# POLITECNICO DI TORINO

Master's Degree in Ingegneria Matematica


Master's Degree Thesis

# Single-solution Based Metaheuristic Algorithms for Capacitated Vehicle Routing Problems: A Comparative Analysis


| Supervisors | Candidate |
| --- | --- |
| Prof. Paolo BRANDIMARTE | Giuseppe Biagio LAPADULA |


March 2024

# Summary

*Vehicle Routing Problems* (VRP) represent a widely studied class of *combinatorial optimization* problems due to their applicability in various real-world contexts, particularly in commercial settings. The inherent complexity of VRP necessitates the use of computationally efficient approximation approaches, with *metaheuristics* emerging as prominent methods in this domain. This thesis aims to construct and develop several metaheuristic-based algorithms, primarily focusing on single-solution approaches, for addressing the *Capacitated Vehicle Routing Problem* (CVRP). Subsequently, these algorithms will be systematically compared against each other and against a state-of-the-art tools (Google Or-tools). Through this comparative study, insights into the efficacy and performance of different metaheuristic strategies in solving CVRP will be gained, contributing to the advancement of optimization methodologies in logistics and transportation management.

# Acknowledgements

ACKNOWLEDGMENTS

*Sarò breve: ringrazio tutti. Tutti coloro a cui voglio bene, tutti coloro che mi sono stati vicino, tutti coloro che ci sono e anche chi non c'è più. Tutti coloro che mi hanno fatto crescere, nel bene, nel male e mi hanno reso la persona che sono oggi. senza il contributo di ognuno di voi, non sarei arrivato al traguardo. Un ringraziamento ed un augurio lo faccio a me stesso, che possa affrontare tutte le sfide, con la mia forza, con tutti voi, Insieme.*

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

Vehicle Routing Problems (VRPs) have increasingly become a class of problems studied within combinatorial optimization, as the volumes of goods movement have continued to rise with the advent of globalization. Consequently, the optimization of freight transport has always been at the forefront of operations research themes, particularly within combinatorial optimization. Subsequently, these types of problems have been adapted to various domains, further emphasizing their fundamental application. The primary challenge that all VRPs face is their computational complexity, leading researchers to avoid traditional "exact" solving approaches used in optimization problems. Instead, commonly utilized approaches involve metaheuristics, algorithms employing approximate methods to reach problem solutions. This thesis will specifically focus on Single Solution Based metaheuristics, with the aim of creating algorithms capable of constructing metaheuristics based on this paradigm. The work will primarily analyze the problem we intend to solve, the Capacitated Vehicle Routing Problem (CVRP), and briefly touch upon other VRP variants. Subsequently, an overview of solution approaches based on metaheuristics will be provided. The following chapters will delve into our work: initially, describing the algorithms produced, followed by conducting tests and analyses.

# Chapter 2

# Problem Description

In this chapter we are going to give some details on the problem that we want to solve. Vehicle routing problem(VRP) is a large class of problems in the field of combinatorial optimization. They aims to find the optimal routes on a map of points. There exist a large number of different kind of VRP variants, but them all are characterized by the fact to have as objective a set of optimal routes to be generated and by the fact that they belong to the NP-hard class of complexity, so they can't be solved in polinomial time.

## 2.1 Travelling Salesman Problem

The *Traveling Salesman Problem* (TSP) originated in the 19th century through the work of Irish mathematician William Rowan Hamilton and British mathematician Thomas Kirkman, who developed Hamilton's icosian game as a recreational pursuit centered on identifying a Hamiltonian cycle. It wasn't until the 1930s, notably in Vienna and at Harvard, that mathematicians like Karl Menger began to delve into the TSP's general form, examining its complexities, including the limitations of straightforward algorithms like the nearest neighbor heuristic.

The TSP stands as one of the earliest incarnations of Vehicle Routing Problems (VRP). It involves navigating a map with a defined set of destinations, with the goal of determining the most efficient route that visits each point exactly once, minimizing overall travel distance. For these reasons, TSP can be represented as an undirected weighted graph, where the cities correspond to the graph's vertices, the routes between cities represent the graph's edges, and the length of each route is considered the weight of the corresponding edge. This problem aims to minimize the total distance traveled, starting and ending at a designated vertex while passing through each other vertex exactly once. Typically, the graph is depicted as complete, meaning there is a direct connection (edge) between every pair of vertices (cities).

If there is no direct route between two cities, it's possible to add an edge with a sufficiently large weight to create a complete graph without affecting the optimal tour.

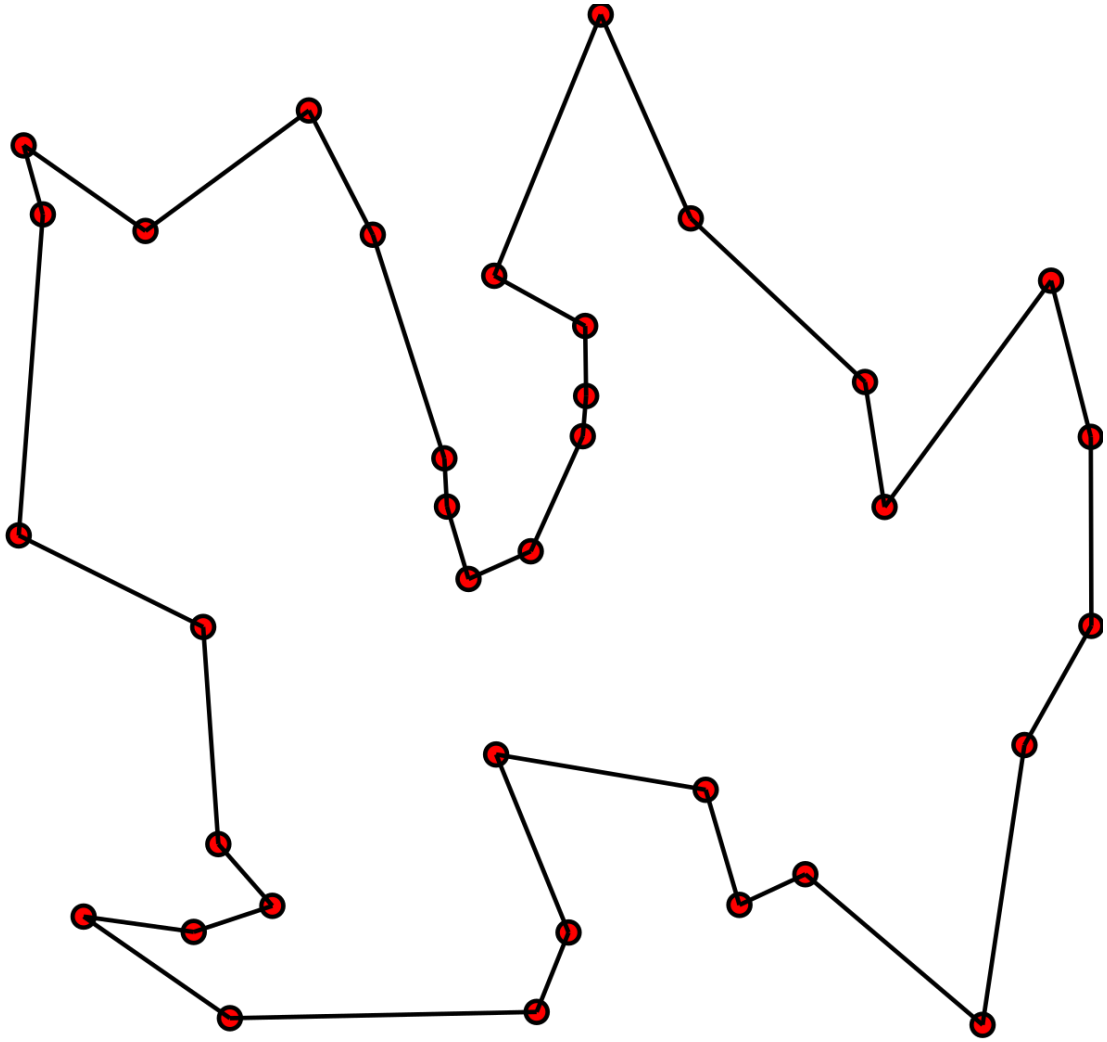

**Figure 2.1:** This is a typical example of a possible solution for this instance of TSP[**wikicommons_example**]

From this ideas a lot of different versions and generalizations of the TSP were created. In particular, we are going to explore one of the most studied version of VRP, the *Capacitated Vehicle Routing Problem* (CVRP).

## 2.2 Capacitated VRP

CVRP is a generalization of TSP, where we insert other issues:

- Now we have to generate more routes.
- Every point of the map represents a client with a certain product quantity demand.
- Every vehicle for every route in the problem has a maximum transport capacity that can't be exceeded by the products.

### 2.2.1 Mathematical Formulation

Our problem consists in a map of points represented by a complete graph $G = (V, E)$, where $|V| = n$. We can find out the formulation of CVRP, given by

$$\text{minimize} \quad \sum_{k=1}^{K}\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ijk}$$

subject to

$$\sum_{k=1}^{K}\sum_{j=1}^{n} x_{ijk} = 1 \qquad \text{for } i = 2,\ldots,n$$

$$\sum_{j=1}^{n} x_{ijk} = \sum_{j=1}^{n} x_{jhk} \qquad \text{for } i,h = 1,\ldots,n, \text{ and } k = 1,\ldots,K$$

$$\sum_{i=1}^{n} d_i x_{ijk} \leq Q_k \qquad \text{for } k = 1,\ldots,K$$

$$\sum_{j=1}^{n} x_{1jk} = 1 \qquad \text{for } i = 2,\ldots,n, \text{ and } k = 1,\ldots,K$$

$$\sum_{j=1}^{n} x_{i1k} = 1 \qquad \text{for } i = 2,\ldots,n, \text{ and } k = 1,\ldots,K$$

$$x_{ijk} \in \{0,1\} \qquad \text{for } i,j = 1,\ldots,n, \text{ and } k = 1,\ldots,K$$

Where:

$n$ : number of nodes (customers plus depot)
$K$ : number of routes (vehicles)
$c_{ij}$ : cost associated with the arc from node $i$ to node $j$
$x_{ijk}$ : binary variable indicating whether there is an arc from node $i$ to node $j$ in route $k$
$d_i$ : demand of customer $i$
$Q$ : maximum vehicle capacity

This model minimizes the total cost of routes while satisfying all customer demands and respecting the maximum vehicle capacities.

One of these points is the depot, the point from which all routes start. Every edge connecting nodes $i$ and $j$ in $V$ has a cost $c_{ij}$. For each node in the graph, there is a specific demand $d_{ij}$, and for every generated route, there is a designated capacity $Q$. Regarding this latter point, there are different considerations regarding capacity. In our case, we assume all vehicles have the same capacity, but this can be generalized to accommodate different capacities for the vehicles.

Furthermore, we can also discuss the number of routes. Classical versions of the problem entail a fixed number of routes to be determined in order to find the optimal solution for these routes. However, in our scenario, we aim to determine the best solution with an optimal number of routes, determined by the solver. Thus, the formulation changes in a manner that allows for the generation of the maximum number of routes, but many of these routes may be "fake" routes as they are empty. Specifically, we have a maximum number of possible routes equal to $n$ (we cannot have more, as this choice assigns exactly one point for every route, and removing a point from a route results in the deletion of the entire route). Clearly, the inclusion of "fake" routes is permissible by modifying the constraints to allow for zero values in certain routes.

With these considerations, we make the final version of the model:

$$\text{minimize} \quad \sum_{k=1}^{K}\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ijk}$$

subject to

$$\sum_{k=1}^{K}\sum_{j=1}^{n} x_{ijk} \leq 1 \qquad \text{for } i = 2,\ldots,n$$

$$\sum_{j=1}^{n} x_{ijk} = \sum_{j=1}^{n} x_{jhk} \qquad \text{for } i,h = 1,\ldots,n, \text{ and } k = 1,\ldots,n$$

$$\sum_{i=1}^{n} d_i x_{ijk} \leq Q \qquad \text{for } k = 1,\ldots,n$$

$$\sum_{j=1}^{n} x_{1jk} \leq 1 \qquad \text{for } i = 2,\ldots,n, \text{ and } k = 1,\ldots,n$$

$$\sum_{j=1}^{n} x_{i1k} \leq 1 \qquad \text{for } i = 2,\ldots,n, \text{ and } k = 1,\ldots,n$$

$$x_{ijk} \in \{0,1\} \qquad \text{for } i,j = 1,\ldots,n, \text{ and } k = 1,\ldots,n$$

Where:

$n$ : number of nodes (customers plus depot)
$c_{ij}$ : cost associated with the arc from node $i$ to node $j$
$x_{ijk}$ : binary variable indicating whether there is an arc from node $i$ to node $j$ in route $k$
$d_i$ : demand of customer $i$
$Q$ : maximum vehicle capacity

We can observe that some constraints that were previously equalities in the general version have now become inequalities ($\leq$), as we may encounter empty routes.

## 2.3 VRP variants

The model shown above is the case that we are going to solve. But in the state of the art, there are a lot of different other cases of VRP problems and variants that we can encounter, changing various aspects, for example adding constraints or changing the objective function that are all linked to different aspects we want to model.

We can see some examples (we can find some of them in [**Elatar_2023**]):

- **Time constraints**: these constraints are used to model the time interval where the vehicle can pass to a certain destination. This leads to the *Capacitated Vehicle Routing Problem with Time Windows* (CVRPTW).
- **Customers returning product**: In the *VRP with Back-hauls* (VRPB), goods need to be delivered to Line-haul customers while other products must be picked up from Back-haul customers.
- **Occasional Drivers**: there can also be some vehicles that are only temporary in *VRP with Occasional Drivers* (VRPOD). [**icores20**]
- **Picking possibility**: The VRP with pick-up and delivery (VRPPD) shares similarities with the VRPB, but in this scenario, each customer may both place orders for certain products and request the pick-up of others.
- **Environmental and pollution Regulation**: with *Green VRP* (GVRP) we have a class of problems where there are some constraints or objectives that model some environmental aspects, such as pollution level, fuel consumption, and other.
- **More depots**: our vehicles can start from multiple locations for their route, as in the *Multi-depot VRP* (MDVRP).
- **Planned horizons for routes**: in some cases, we may need to model the routes to be used more than one time, as in *Periodic VRP* (PVRP).
- **Drones**: there are particular variants of VRP where we have to manage drones, for example in [**Shi_2023**] the *UAV routing* aims to solve a problem where the drones have limited battery and they can go to different charge stations; for that, there are some constraints that manage this aspect.

# Chapter 3

# Solution Approaches

As we sayed in the previous chapter, VRPs are NP-hard problems and for this reason solving them is a very challenging issue when the dimension of the problem becomes higher and higher(curse of dimensionality): it isn't easy use conventional exact solvers, that result heavy and uneffective for the computational issues. In front of these problems, a class of different minimizers have been created; these use some strategies defined *"heuristics"*, that are algorithmic strategies(taken from different kind of ideas and concepts, as we will see later) that can give us an approximated result with a certain level of quality, but with good computational properties. In the case of VRPs we can consider two kind of heuristics [**liu2023heuristics**]:

- Constructive heuristics: Algorithms in this category build routing solutions from scratch following fixed empirical heuristic procedures. They typically generate a feasible solution quickly and are easy to implement across various VRP variants. However, solutions produced by constructive heuristics often exhibit a certain gap from the optimal solution
- Improvement heuristics: These iteratively enhance an incumbent routing solution by conducting a local search in the neighborhood. They are quite efficient at determining a local optimum. The main limitation is that they can easily become trapped in local optima, and the final solution's quality depends on the starting point of the local search

From these tricks, we can get algorithms that exploits and put togheter them all in order to get some solvers that points to get good approximated solutions in brief times, in a more general way, not too much linked to the single problem but more generalizable. These are called *"meta-heuristics"*. As we can see in [**liu2023heuristics**]:

- Metaheuristics differ from constructive heuristics and improvement heuristics in the fact that they focus on high-level algorithm principles rather than exploiting

the specific features and structure of the problem. Instead, metaheuristics draw inspiration from natural phenomena or physical processes to design optimization algorithm paradigms. They are generally less dependent on the problem at hand and are known for their efficiency and global search capabilities.

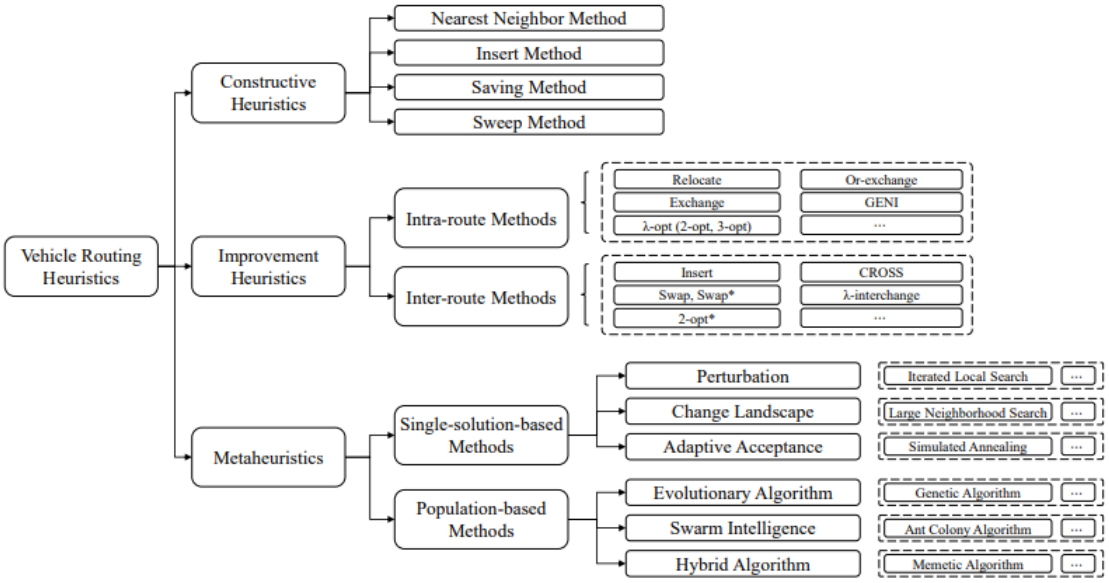


**Figure 3.1:** This map summarizes the various kinds of heuristic approaches in [**liu2023heuristics**]

## 3.1 Constructive Heuristics

Constructive heuristics are algorithms that create routing solutions from scratch based on predefined empirical procedures. While they are able to generate solutions quickly, there is typically a gap between their results and the optimal solution. Existing constructive heuristics can be categorized into four main algorithm frameworks: the nearest neighbor method, the insert method, the saving method, and the sweep method.

- **Nearest neighbor method**: the nearest neighbor method is a simple constructive heuristic often used for solving VRPs. This method builds routes by greedily adding the nearest feasible unrouted customer to each route, starting from the depot. Routes can be built sequentially or in parallel. In sequential route building, routes are extended one at a time, with a new route initiated

from the depot when no more customers can be added. However, this approach may lead to uneven loading among routes, especially for the last vehicle. To alleviate this issue, parallel route building is employed, where a predetermined number of vehicles are used, and routes are extended simultaneously. Each iteration adds the closest unrouted customer to each route, ensuring that K customers are added in total. This process continues iteratively until all customers are visited. Although the nearest neighbor method has a time complexity of $O(n^2)$ with n customers, it has been integrated into various VRP algorithms in recent years. It serves as an initialization technique in meta-heuristic algorithms like tabu search, simulated annealing, memetic algorithm, and large neighborhood search, contributing to their effectiveness.

- **Insert method:**the insert method initializes empty routes and inserts unrouted customers into them one by one, allowing for insertion at positions other than the end of each route as in the nearest neighbor method. The insertion with the minimum cost is performed in each iteration. This method works by calculating the cost of inserting an unrouted customer into a route and selecting the insertion position with the lowest cost. This process is repeated for each unrouted customer. The worst time complexity of the insert method is $O(n^3)$, as it iterates through all customers and calculates insertion costs for each. Variations of the insert method include the farthest insert and regret insert. The farthest insert selects the farthest unassigned customer for insertion in each iteration, while the regret insert extends the greedy insert by considering regret, which is the difference in cost between the best and alternative insertion positions. Hybrid implementations of insert methods have been developed to improve performance. These include combining sequential and parallel procedures for route construction, using insert heuristics in conjunction with other algorithms such as ant colony optimization, and implicitly employing insert methods as recreate operations in large neighborhood search algorithms.

- **Saving method**: the saving method, introduced in 1964 by Clarke and Wright [**Clarke_1964**], is one of the most well-known constructive heuristics for solving VRPs. It begins with an initial solution where each customer is served by a separate route, and then iteratively merges shorter routes into longer ones to reduce overall costs. This merging process can be performed either in parallel or sequentially. In the parallel version, the method combines two routes with endpoints i and j, producing the maximum feasible distance saving. In the sequential version, each route is considered individually, and feasible merging operations are applied iteratively until no further improvements can be made. The time complexity of a straightforward implementation of the saving method is $O(n^3)$, where n is the number of routes. However,

this complexity can be reduced by precomputing and sorting possible route combinations at the beginning of the algorithm. Early revisions of the saving method involved parameterizing the saving equation and considering only the first-pair calculated savings. Additionally, the saving method has been combined with other concepts, such as insert methods and matching-based algorithms, to produce better results with higher computational costs. More recently, the saving method has been extended and refined by various methods, including divide-and-conquer procedures, genetic algorithms, and stochastic versions of the classic heuristic. These advancements have enabled the saving method to be applied to a wide range of VRP variants and real-world problems, demonstrating its simplicity and flexibility in implementation.

- **Sweep method**: The sweep method, introduced in seminal work [**Gillett_1974**], involves sorting nodes according to polar angle and adding them to routes in a circular manner starting from the depot. If insertion is infeasible, a new route is created. An alternative approach clusters customers based on polar angle and solves a TSP within each cluster. Performance can be affected by the depot's location, with off-centered depots leading to poorer results. Various reference points can mitigate this issue. The basic sweep method's time complexity is $O(n)$, but advanced versions depend on the TSP-solving method. Two straightforward implementations of the sweep method are the sweep nearest algorithm and distance-based sweep nearest algorithm. These algorithms select the next nearest customer in each cluster, outperforming the basic sweep method and rivaling modern heuristics. The sweep method has been extended to various VRP variants, including capacitated VRP, mix vehicle routing problems. These extensions leverage sweep heuristics combined with local search, integer programming, and genetic algorithms to improve solution quality and efficiency, as we will see later.

## 3.2 Improvement Heuristics

Improvement heuristics delve into the vicinity of the current solution to achieve objective enhancements. They have the capability to swiftly converge towards local optima, making them effective for addressing extensive routing challenges. These heuristics are generally classified into two distinct categories: intra-route and inter-route. The distinction lies in their neighborhood structures. Intra-route heuristics explore within a single route, whereas inter-route heuristics involve multiple routes.

- **Intra-route heuristics**: intra-route improvement heuristics focus on exploring the neighborhood within a single route. Many of these techniques have their origins in local search operators developed for the TSP. For example,

simple operations involve relocating a single customer to a different position within the same route or exchanging the positions of two customers within a route. One of the more versatile intra-route heuristics is the $\lambda$-opt heuristic, which operates on edges. It involves removing $\lambda$ edges from a route and then recreating $\lambda$ to connect disjoint sequences. Given that a full implementation of $\lambda$-opt moves necessitates $O(n^{\lambda})$ operations for n customers, smaller values of $\lambda$ such as *2-opt*, *3-opt*, and *Or-exchange* are commonly used. In the next image it's possible to see some examples of intra-route methods and their applications.

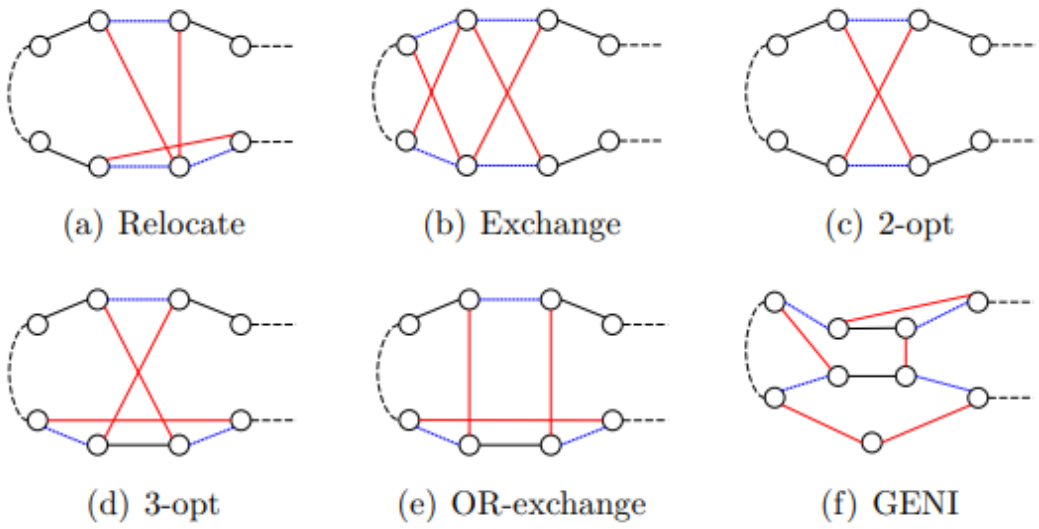


**Figure 3.2:** Various intra-route heuristics applications on a route [**liu2023heuristics**]

- **Inter-route heuristics**: inter-route heuristics involve local searches spanning multiple routes, often building upon intra-route counterparts. For instance, *insert* and *swap* operations extend *relocate* and *exchange* operations, respectively. *Insert* removes a customer from one route and inserts it into another, while *swap* exchanges two customers from different routes. Additionally, the exchange of two edges from different routes is termed *2-opt** to distinguish it from the traditional *2-opt* method. *CROSS* swaps two strings, each containing at most $\lambda$ customers. $\lambda$-interchange further expands upon *CROSS* by permitting the exchange of any set of fewer than $\lambda$ nodes between two routes, even if they are not consecutive.

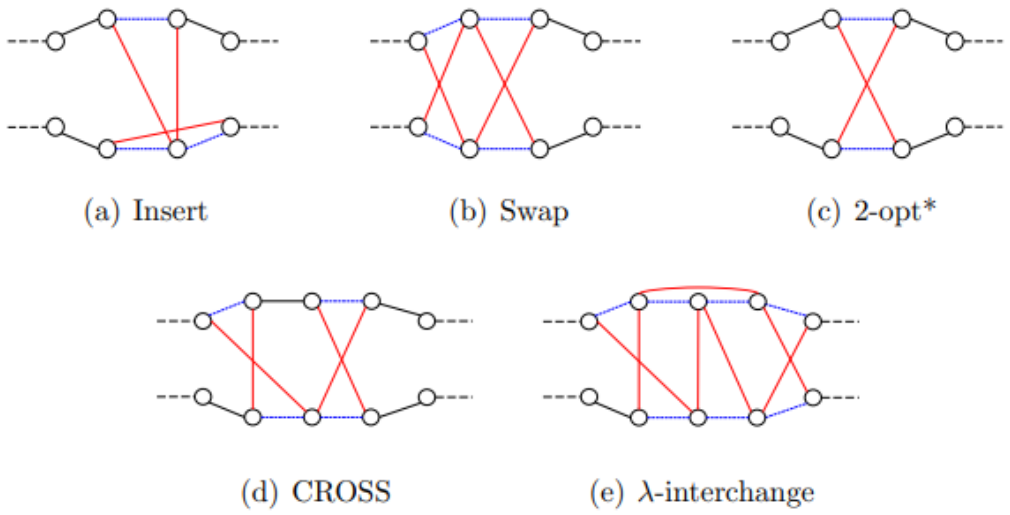


**Figure 3.3:** Various inter-route heuristics applications between two different routes [**liu2023heuristics**]

About improvement heuristics, in the next chapter we will talk more specifically of some cases that we will use for our algorithms.

## 3.3 Metaheuristics

Metaheuristics represent a higher-level approach compared to constructive and improvement heuristics, as they are problem-independent and aim to tackle hard optimization problems more generally. Widely regarded as effective for various optimization challenges, metaheuristics are gaining prominence in vehicle routing research due to their efficiency and scalability. These methods can be categorized based on their population management strategy into single-solution-based and population-based methods. Single-solution-based methods, also known as neighborhood-based or local search-based methods, iteratively perform low-level searches on a single incumbent solution, typically employing improvement heuristics. High-level rules guide the search process to escape local optima and explore the search space effectively. Strategies for designing these rules include multiple starts, changing the landscape, and designing acceptance rules. Population-based searches, on the other hand, involve approaches like evolutionary algorithms and swarm intelligence. Evolutionary algorithms, such as genetic algorithms, are inspired by biological evolution, where individuals better suited to the environment are more

likely to produce offspring with advantageous traits. Swarm intelligence examines the cooperative behavior of decentralized systems, drawing inspiration from social entities like bee colonies and ant colonies. In vehicle routing, widely-used metaheuristics include *simulated annealing, tabu search, iterated local search, large neighborhood search, variable neighborhood search, genetic algorithm, ant colony optimization*, and *memetic algorithm*. These algorithms cover various sub-classes of metaheuristics and are among the most frequently used in vehicle routing research.

### 3.3.1 Single-Solution-Based Methods

- **Simulated annealing**: *Simulated annealing* (SA) originated in the early 1980s as a method for combinatorial optimization, inspired by the physical process of annealing in materials science. In SA, an incumbent solution is mutated using local search operators to explore the solution space. The fitness of the solution corresponds to its energy level, and the algorithm aims to find the state with the lowest energy, analogous to reaching a stable state in annealing.The acceptance of new solutions in SA follows a probabilistic criterion:

  - If the new solution improve the objective function value, take the solution.
  - If the new solution is worst than the preceeding solution, we have a probability equal to $\frac{\exp(f(s_{new})-f(s_{old}))}{T}$, where $T$ is parameter of the algorithm, witch inizialization starts with it. Value of $T$ changes during the run,

  As the temperature decreases over iterations, the probability of accepting worse solutions decreases, leading the algorithm to converge towards the optimal solution while avoiding getting trapped in local optima. In vehicle routing, SA was first applied in the early 1990s, where it was hybridized with tabu search for solving CVRP [**Osman_1993**]. Since then, SA has been utilized in various VRP variants, including those with time-window and more. Recent advancements in SA for VRP include adaptive mechanisms for selecting neighborhood moves and population-based approaches, where a population of solutions is improved using SA heuristics and crossover operators.Overall, SA offers a versatile and effective approach for tackling complex optimization problems like VRP, providing researchers with a valuable tool for finding high-quality solutions.

- **Taboo Search**: *Tabu search*, introduced in 1986 [**Glover_1986**], offers a systematic approach to circumvent the issue of local optima in search algorithms and guide the exploration towards promising directions. A key feature of tabu search is the tabu lists, which record recent search history and prevent revisiting previously explored solutions. In the realm of vehicle routing,

early applications of tabu search were explored in various works. Subsequent enhancements to the basic framework have been proposed, such as adaptive memory programming, which leverages a set of memory components for more efficient search management. Granular tabu search [**Toth_2003**], introduced to vehicle routing, restricts the neighborhood to prioritize "short" arcs, based on the observation that shorter arcs are more likely to contribute to high-quality solutions. This concept has gained popularity and been integrated into other local search-based methods. The applications of tabu search extend across various VRP variants, including multi-depot VRP, periodic VRP, scenarios involving discrete split deliveries and pickups and others.

- **Iterated Local Search**: *Iterated local search* (ILS) operates on the principle of iteratively generating a sequence of solutions using underlying heuristics. Unlike simulated annealing (SA) and tabu search (TS), ILS employs perturbation to escape local optima instead of modifying fitness or acceptance criteria. Perturbation methods range from random restarts to more structured strategies, often leveraging various neighborhood search heuristics.ILS finds application across a wide range of VRP variants, including time-dependent VRP, VRP with backhauls, and others. Recent enhancements to the basic ILS framework include memory-based ILS[**Brand_o_2020**], which utilizes optimization history for defining perturbation procedures, and population-based ILS [**Sabar_2022**], which maintains a population of promising solutions and employs evolutionary operators for dynamic VRP. Additionally, adaptive ILS with diversity control methods has shown competitive results, particularly for large-scale problems.

- **Large Neighborhood Search**: *Large Neighborhood Search* (LNS) (seen in [**Shaw1997ANL**]) capitalizes on the concept that a larger neighborhood increases the likelihood of containing high-quality local optima. While any neighborhood structure can be employed in the LNS framework, for vehicle routing problems, it typically involves two primary procedures: *ruin and recreate*, also known as *destroy and repair*. Ruin entails removing a portion of the current solution, while recreate reinserts the removed portion to form a new solution. The ruin phase often considers the interrelatedness of removed customers, typically measured by factors like distance and other similarities. The recreate phase commonly employs constructive heuristics, such as insert methods, to reinsert the removed portion. Adaptive Large Neighborhood Search (ALNS), an extension of LNS, as discussed in [**Ropke_2006**], incorporates multiple ruin and recreate operators and dynamically selects these operators in each iteration. The selection probability for each operator is adjusted based on its historical performance during optimization, leading to enhanced adaptability and performance. Conversely, in [**Christiaens_2020**]

proposed an LNS heuristic without multiple operators or adaptive weighting. This approach utilizes adjacent string removal for ruin and a greedy insert with blinks for recreate, demonstrating superior performance compared to other state-of-the-art methods. This study highlights that simplicity and reproducibility can sometimes yield competitive solutions without sacrificing quality. Anyway, this concept to ruin and recreate the solution will be used in the next chapter.

- **Variable Neighborhood Search**: *Variable Neighborhood Search* (VNS) method, formally introduced in [**Mladenovi___1997**]. In each iteration of a VNS-based heuristic, the following three steps are executed sequentially until a termination condition is met:

  1. Shaking Procedure: Introduces randomness or perturbations into the current solution.
  2. Improvement Procedure: Applies local search to enhance the solution quality.
  3. Neighborhood Change: Shifts to a different neighborhood structure to explore diversified solution spaces.

  This iterative process requires an initial feasible solution, typically randomly chosen, from which iterations commence. VNS leverages the understanding that local minima differ across various neighborhood structures, underscoring the importance of exploring multiple neighborhoods to avoid being trapped in suboptimal solutions. By simplifying the approach and requiring minimal parameters, VNS not only yields competitive solutions but also provides insights into its performance, facilitating the development of more efficient implementations. At the core of VNS lies the neighborhood change function, which evaluates improvements in solution quality across different neighborhoods. Depending on the problem at hand, VNS can operate deterministically, stochastically, or through a combination of both approaches. Extensions to the basic VNS framework include variants such as descent-ascent methods, first-improvement methods, and strategies involving multiple local searches from randomly generated solutions. Each variant offers unique advantages, contributing to the versatility and adaptability of VNS in solving optimization problems. For instance, in solving VRP, such as the VRPTW [**Macrina_2020**] [**Ferreira_2018**], VNS is widely utilized. By providing efficient and effective solutions, VNS has established itself as a prominent metaheuristic approach, offering a robust framework for tackling complex optimization challenges.

### 3.3.2 Population-Based Methods

- **Genetic Algorithms**: *Genetic algorithms* (GA) have been a prevalent optimization method for decades, drawing inspiration from natural evolution to maintain a balance between population diversity and adaptiveness. The idea involves generating a pool of solutions with the aim of enhancing a fitness function, ultimately identifying the most optimal solutions through an evolutionary process that harnesses genetic mechanisms. Key to GA are two fundamental evolutionary operations: crossover and mutation. Mutation involves permutation on a solution to generate new offspring, with various permutation operators considered as types of mutation. Crossover, on the other hand, facilitates an exchange between solutions to produce offspring from two selected parents. Common crossover operators in vehicle routing, inherited from those used in genetic algorithms for the TSP, include Order Crossover, Partially Mapped Crossover, Edge Recombination Crossover, Cycle Crossover, and Alternating Edges Crossover. The application of GA in solving vehicle routing (for instance [**Baker_2003**]) gained traction about twenty years ago . As GA was applied to various modern VRP variants such as multi-depot VRP, pickup and delivery problems, green VRP, and multiobjective VRPTW, new crossover and mutation operators were developed. Recognizing that traditional GA may not be aggressive enough for combinatorial optimization problems, integrating GA with different search techniques has become a popular trend. Integrated methods include combining GA with particle swarm optimization, simulated annealing, and sweep-based techniques to enhance performance on challenging routing problems.

- **Ant Colony Optimization** : *Ant colony optimization* (ACO) draws inspiration from the behavior of real ants, which communicate using pheromones. ACO constructs solutions to optimization problems based on these pheromones, updating them during the search process to reflect search history. The pioneering ant system applied this concept to the TSP. Expanding ACO to VRPs, [**Bell_2004**] introduced a method for searching multiple routes. Building upon this, recent research has focused on combining ACO with other algorithms to tackle challenging VRPs . Applications of ACO include multi-compartment VRP, VRP with simultaneous pickup and delivery, heterogeneous VRP with mixed backhaul, multi-depot green VRP with multiple objectives, multiobjective VRP with flexible time windows, periodic VRP with a time window and service choice, dynamic VRPs, and capacitated electric VRP.

# Chapter 4

# Algorithms Modules

In this section, we will describe the various algorithmic structures we have developed, mainly derived from Single Solution Methods (which we will examine in more detail later). These algorithmic structures can actually be thought of as "modules" that can be inserted into a general algorithm aimed at finding a solution. Therefore, once these "modules" are described, we will proceed to describe some Single Solution-based metaheuristics constructed by leveraging them. All the algorithms are written using **python** language.

## 4.1 Modules

### 4.1.1 Inspirations

The algorithms that we drew inspiration from to build these modular structures come, as mentioned earlier, from other well-known metaheuristics. In particular, the main references were the *Variable Neighborhood Search* (VNS) and the *Iterated Local Search* (ILS). Additionally, other borrowed structures come from different paradigms such as *Large Neighborhood Search* (LNS), *Differential Evolution* (DE) (the only structure that arises from "Population Based" metaheuristics) and a little from other.

### 4.1.2 Improvement Heuristics

Since our algorithms are primarily based on local search structures, the first thing to discuss is the local search structures we have developed. Local search algorithms aim to explore at every iteration the solutions near to the current explored solution and these are called *Neighborhood* . We should note that, given the considered problem (CVRP), we wanted to choose a "geometrically" intuitive way to represent the solutions. Indeed, our final solution is a set of routes that start from a depot

and return to it. Therefore, for search and representation purposes, the solutions are represented with lists of arrays of integers (routes), where the first and last value is 0. The integers contained in each individual route correspond to indices referring to the position of the explored destination; the indices are assigned to the destinations based on the order of appearance in the matrix of points corresponding to the destinations, whose index 0 indicates the position of the depot.

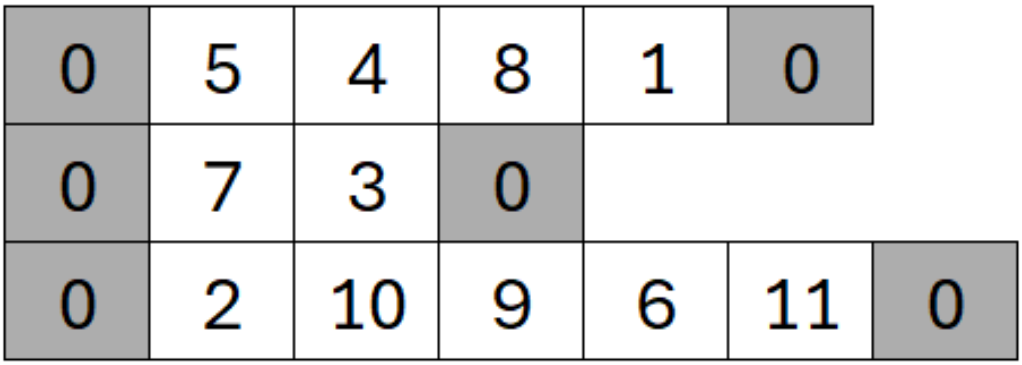


**Figure 4.1:** An example of three routes structure representation in our work: grey rectangles with 0 represent the depots, while the other numbers are the indexes of nodes where we have to pass in a single route.

So, local heuristic algorithms will essentially be variations of the geometric structure of the routes (provided they respect the constraints), and they can employ various strategies. In particular, we have developed several structures and their generalizations, which we can summarize based on these "neighborhood movements"(we take inspiration for some structures from [**Macrina_2020**] ):

- Intra route movements:
  - Swapping nodes in a route.
  - Node relocation movements: taking a node and putting it in the best position in the route.
- 2-Opt exchange: splitting a route in 3 parts and reconstruct it with the middle piece inverted in direction.
- Inter route movements:
  - Moving nodes from a route to another.

- Swapping nodes between routes.
- 2-Opt* exchange: taking two routes, splitting them in two pieces and then merge the first part of first route with the last part of the second route and vice-versa.
- Creating new routes taking nodes or edges from the other routes.
- Creating a new route merging two routes that have more exploitable capacity.
- Breaking one route in two route pieces.

In the various developed algorithms we choose between these heuristic what to be used to search locally the solution.

### 4.1.3 Improvement and Shaking algorithms

The heuristics described in the last section have been built in order to be used in the improvement and shaking algorithms, two algorithms belonging to VNS paradigma. These represent two phases of the VNS: shaking phase is used to explore the neighborhoods while the improvement phase is used to do the local search of solution.

Shaking phase can be designed in different ways: we choose to use a randomic approach, that uses to apply on the current solution one o more improvement heuristic functions in order to get new restarting solution for the improvement phase. The basic structure can be seen as in [**2010**]:

**Algorithm 4** Shaking function

**Function** Shake($x, k$)
1 $w \leftarrow [1+\text{Rand}(0,1) \times |\mathcal{N}_k(x)|]$
2 $x' \leftarrow x^w$
**return** $x'$

**Figure 4.2:** [**2010**], chapter about VNS algorithms: the $x$ and $k$ parameters represent respectively the current solution and the neighborhood structure to be used. $N_k(x)$ is referring to the neighborhood function of current solution for the k-th neighborhood structure.

Improvement phases, as shaking phase, applies improvement heuristic algorithms to explore it's neighborhood, but it's ojective is different: it works as a local search engine, that applies heuristics in order to cause the descent of objective function. We can build it in a lot of different ways. In our case, we have chosen to exploit the two standard versions of improvements:

- Best improvement: we explore all our neighborhood in order to find the best solution.

**Algorithm 5** Best improvement (steepest descent) heuristic

**Function** `BestImprovement(x)`

```
repeat
    x' ← x
    x ← arg min_{y∈N(x)} f(y)
until (f(x) ≥ f(x'))
return x
```

**Figure 4.3:** [2010]

- First improvement: at this iteration we accept the first solution that descend our objective function value.

**Algorithm 6** First improvement (first descent) heuristic

**Function** `FirstImprovement(x)`

```
repeat
    x' ← x;  i ← 0
    repeat
        i ← i + 1
        x ← arg min{f(x), f(x^i)},  x^i ∈ N(x)
    until (f(x) < f(x') or i = |N(x)|)
until (f(x) ≥ f(x'))
return x
```

**Figure 4.4:** [2010]

As we can see, First Improvement can be a faster algorithm compared to Best Improvement because it has to explore less neighbours, but it doesn't find the best neighbour in the improvement running, so it could need more application then best improvement in order to find the best solution.

Another function used in VNS is the *Neighborhood change* algorithm, that allows to change the neighborhood structure at each iteration. In this case, we choose tho not deploy this function apart of the other, but integrate it into both shaking and improvement phases.

Now, we will show some examples of shake and improvement algorithm developed by us, exploiting also the fact to integrate in them the neighborhood change

**Algorithm 1** Shake

**Input:** $sol, routes, points, demands, Q, probs$

**Data:** $eps = 0.05$; *structs* array of indices of neighborhood structures to pass to `neighborhood`;

**if** *length of probs* $= 0$ **then**
  $probs \leftarrow$ an array of length $len(structs)$ with all values equal to $\frac{1}{\text{length of structs}}$;
**end**
$neigh_struct \leftarrow$ randomly select an element from *structs* with probability
$(routes, difference) \leftarrow$ apply the `neighbor` function with parameters $neigh_struct$, *routes*, *points*, *demands*, $Q$;
$N \leftarrow$ randomly select an integer between 2 and 30;
$i \leftarrow 0$;
**while** $i < N$ **do**
  $neigh_struct \leftarrow$ randomly select an element from *structs* with probability *probs*;
  $(routes, difference) \leftarrow$ apply the `neighbor` function with parameters: *neigh_struct*, *routes*, *points*, *demands*, $Q$
  **if** $difference < 0$ **then**
    $neg\_pos \leftarrow$ find the index of *neigh_struct* in *structs*
    $p \leftarrow probs[neg\_pos]$
    $restarter \leftarrow p < \frac{1}{1+eps}$
    **if** *not restarter* **then**
      $probs[neg\_pos] \leftarrow \min((1 + eps) \cdot p, 1)$
      $\beta \leftarrow \frac{(1-(1-eps)\cdot p)}{(1-p)}$ Update the values of *probs* except for the index *neigh_struct* according to the rule $\beta$
    **end**
  **end**
  $sol \leftarrow sol + difference$
  $i \leftarrow i + 1$
**end**

**Output:** $routes, sol, probs$

The shake function in the algorithms selects one of the possible neighbors contained in the structure *struct* randomly using a probability vector *probs*. In the case of a solution descent, the neighborhood structure is updated by increasing the probability of selecting the good neighborhood by an $\varepsilon$ fraction (and adjusting the other probabilities accordingly). This technique is known as adaptive shaking [**Brimberg_2023**]. The "neighborhood" function has to link the shaking procedure with the respective chosen neighbour structure. The inputs of the shaking function $sol, routes, points, demands, Q$ are respectively current value of objective function, current solution of routes, points of the maps, demand vector and capacity for

every route. For improvement I will show you two variants. You can build a lot of different possibilities. We will show you two of our improvement structural ideas:

**Algorithm 2** improvement

**Input:** $sol, routes, points, demands, Q, hmax, first$
**Data:** $n\_str$ indici strutture di vicinato da esplorare;
$new\_sol \leftarrow$ una copia di $sol$
$best\_solution \leftarrow$ una copia di $sol$
$best\_routes \leftarrow$ una copia di $routes$
$h \leftarrow 0$

```
1  foreach neigh_struct in n_str do
2      while h < hmax do
3          new_routes, difference ← neighbour(neigh_struct, routes, points, demands, Q)
4          if difference < 0 then
5              new_sol ← sol + difference
6              if new_sol < best_solution then
7                  best_solution ← una copia di new_sol
8                  best_routes ← una copia di new_routes
9                  if first then
10                     break
11                 end
12             end
13         end
14         h ← h + 1
15     end
16 end
17 feasible, best_routes, _ ← constraints(best_routes, demands, Q)
18 if feasible then
       Output: best_routes, best_solution
19
20 end
```

In this case we can see that we are using only one kind of local solution search, based on the neighborood in $n\_str$. Let's see another case, that splits the search in two parts:

**Algorithm 3** improvement

**Input:** $sol, routes, points, demands, Q, hmax, first$

**Data:** $diversification, intensification$ array di indici delle strutture di vicinato per diversificazione e intensificazione $best\_solution \leftarrow$ una copia di $sol$ $new\_sol \leftarrow$ una copia di $sol$ $best\_routes \leftarrow$ una copia di $routes$

```
1  foreach neigh_struct in diversification do
2      h ← 0
3      while h < hmax do
4          (new_routes, difference) ← neighbour(neigh_struct, routes, points, demands, Q)
5          if difference < 0 then
6              new_sol ← sol + difference
7              if new_sol < best_solution then
8                  best_solution ← una copia di new_sol
9                  best_routes ← una copia di new_routes
10                 if first then
11                     break
12                 end
13             end
14         end
15         h ← h + 1
16     end
17 end
18 d_routes ← una copia di best_routes  new_sol ← una copia di best_solution
19 foreach neigh_struct in intensification do
20     h ← 0
21     while h < hmax do
22         (new_routes, difference) ← neighbour(neigh_struct, d_routes, points, demands, Q)
           if difference < 0 then
23             new_sol ← diverse_sol + difference
24             if new_sol < best_solution then
25                 best_solution ← una copia di new_sol
26                 best_routes ← una copia di new_routes
27                 if first then
28                     break
29                 end
30             end
31         end
32         h ← h + 1
33     end
34 end
```

**Output:** $best\_routes, best\_solution$ if feasible;

In this case, we can see that there are two phases of improvement: the first phase is called *diversification*: here to improve solution only inter route neighborhood are used, in a way to improve route clusters; the second phase is the *intensification* phase, where are used intra route neighborhood structures, in such a way to get routes shapes of better quality.

In both our improvement algorithms and the shake function, we use a similar method to select neighborhoods, but in different ways. Also, in the improvement algorithms, there's a function to check if the solution works for the problem. Overall, our algorithms calculate the difference between two solutions when exploring the neighborhood. This is easier computationally because we can just find the difference in distance values where the current routes and the new solution are different (computing the difference for entire solution on large instances can be very impacting on computation time). These calculations are part of each neighborhood function. Once we have the difference value, we can find the new solution value by adding it to the old solution.

### 4.1.4 Destruction/Reconstruction

In some contexts, perturbing the solution solely by exploring neighborhoods using the shaking function could be too conservative, and it might not facilitate the exploration of new solutions effectively. In this sense, drawing inspiration from some Large Neighborhood Search algorithms, we have devised a destruction/reconstruction framework to significantly perturb our current solution during iterations. This paradigm can be divided into two main stages:

- *Destruction phase*: we take a solution and, based on certain criteria, remove some points from the routes.
- *Reconstruction phase*: here, we take the destroyed routes and re-add all the missing points following specific rules.

More in detail, we can show the pseudocodes of this two phases:

**Algorithm 4** Destroy

**Input** : $routes, points, demands, Q$
**Output** : $candidateRoutes, toBeRemoved$

```
1  total ← size of the points array;
2  candidateRoutes ← a copy of routes;
3  val ← 0;
4  toBeRemoved ← an empty array;
5  while stopping condition for feasibility do
6  |  candidateRoutes ← a copy of routes;
7  |  movement ← randomly choose a number between 0 and 3
8  |  if movement == 0 then
9  |  |  candidateRoutes, tBR ← random_client_removal(routes, points, demands, Q)
10 |  end
11 |  else if movement == 1 then
12 |  |  candidateRoutes, tBR ← zone_removal(routes, points, demands, Q)
13 |  end
14 |  else if movement == 2 then
15 |  |  candidateRoutes, tBR ← prox_based_removal(routes, points, demands, Q)
16 |  end
17 |  else if movement == 3 then
18 |  |  candidateRoutes, tBR ← random_route_removal(routes, points, demands, Q)
19 |  end
20 end
21 toBeRemoved ← convert tBR into an array of integers;
22 return candidateRoutes, toBeRemoved;
```

As you can see in the destroy algorithm, we have 4 ways to destroy our routes, we take inspirations from [**Shi_2023**]:

- *random client removal*: it chooses random clients to be removed from the routes.
- *zone removal*: it chooses one client, simulates a rectangular zone around it, simulating $x$ and $y$ axis extension following a normal distribution based on the distances of all points from the center(calculating their sampling mean and sampling variance); after this, it selects some points from those contained in the the zone.
- *prox_based_removal*: similar to random removal, it allows to remove randomly some edges(couples of points one next to the other in the routes).
- *random_route_removal*: it removes a randomly chosen entire route.

These algorithms give in output modified routes and a list of the removed nodes.

**Algorithm 5** repair

**Input** : $removed, routes, points, demands, Q$
**Output** : Repaired routes $routesMod$

```
1  total ← size of points
2  remotion ← copy of removed
3  feasible ← true
4  routes_trunk ← list of middle parts of routes
5  monoRoute ← concatenation of routes_trunk
6  routesMod ← copy of routes
7  remotionMod ← copy of remotion
8  while size of remotionMod > 0 and feasible do
9      movement ← random choice from the insertion structures indexes
10     inputStruct ← [remotionMod, routesMod, points, demands, Q]
11     switch movement do
12         case 0 do
13             routesNew, remotion ← greedy_ insertion(inputStruct)
14         end
15         case 1 do
16             routesNew, remotion ← fastGreedy_ insertion(inputStruct)
17         end
18         case 2 do
19             routesNew, remotion ← random_ insertion(inputStruct)
20         end
21         case 3 do
22             routesNew, remotion ← randomGreedy_ insertion(inputStruct)
23         end
24         case 4 do
25             routesNew, remotion ← newRoute insertion(inputStruct)
26         end
27     end
28     if feasibility conditions then
29         routesMod ← copy of routesNew
30         remotionMod ← copy of remotion
31     end
32 end
33 return routesMod
```

As is possible to see, reconstruction phases applies reconstruction operators until the remaining points vector has no more points, so the new routes are full. These reconstruction operators are (some of them comes from [**Shi_2023**] ):

- *greedy_ insertion* : the operator computes the insertion cost for each removed destination in the best insertion position, and iteratively inserts the node with the lowest insertion cost into its feasible positions within the current solution.
- *fast_greedy_insertion* : similar to greedy_insertion more randomical.
- *random_ insertion* : totally random insertion.
- *random_greedy_insertion* : it's a version of random_insertion hybridated with greedy insertion
- *newRoute_insertion* : it initializes and inserts points in a new route while they respect the capacity constraint.

In order to get new solutions, the procedures destroy and reconstruct are concatenated.

### 4.1.5 Crossing-over operators

When we perturb the solution to expand the search, could be important also improve the quality of the new starting solution for the local search "hybridizing" it with the best solutions found in past iterations. For this reason, we were inspired by a Differential Evolution algorithm in [**Souza_2023**] to create an operator that applies the crossing-over operator. In our context, we use crossing-over in two ways:

- When we perturb a solution, we use it to improve the quality of perturbation.
- When we get a new solution from improvement, we try to improve it hybridizing it with the other solutions computed before.

The perturbation scheme functions as follows: we start with our main reference solution and another solution that we'll call the test solution. The aim is to randomly select a position within the routes of both solutions. If a corresponding position (route, position within the route) exists in both the main solution and the test solution for the selected position, we swap the node from the main solution with the node found at the corresponding position in the test solution, all within the main solution.

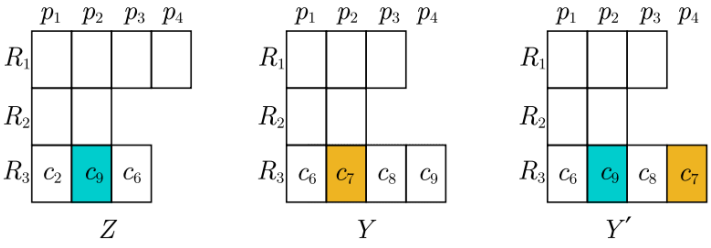


**Figure 4.5:** [**Souza_2023**], in this example $Z$ is the test solution while $Y$ is the main solution: it is selected the position $(route = 3, route\_position = 2)$; in $Z$ it is equal to $c9$, while in $Y$ it is equal to $c7$. Then, we exchange in $Y$ node $c7$ with $c9$, getting a new solution $Y'$

This procedure will be applied for different randomly chosen nodes, with a probability value that allows to randomly choose if apply the crossing node procedure that is given in input.

### 4.1.6 Starting and Restarting Methods

For metaheuristic algorithms, it's also important to have starting points from we start a local search or we restart it( in the case of ILS framework). In order to have starting routes, we have to consider some aspects, based on how much we want to be greedy or random in starting solution generation. In this sense, we have to consider two aspects to be controlled in the routes generation:

- *Clustering strategy*: before we have to divide points between clusters that respect the capacity costraints, and we can clusterize point with different strategies:
  - Totally random approach;
  - Sweep algorithm based: in this case we use the *Sweep Algorithm* heuristic; In particular, this algorithm sorts customers based on their angular position in relation to the central depot. This angle is determined by computing the arctangent of the differences in the x and y coordinates between each customer and the depot, atan2$(y_c - y_d, x_c - x_d)$, where:

$$\text{atan2}(y,x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & \text{se } x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & \text{se } x < 0 \text{ e } y \geq 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & \text{se } x < 0 \text{ e } y < 0 \\ +\frac{\pi}{2} & \text{se } x = 0 \text{ e } y > 0 \\ -\frac{\pi}{2} & \text{se } x = 0 \text{ e } y < 0 \\ \text{undefined} & \text{se } x = 0 \text{ e } y = 0 \end{cases}$$

Consequently, customers are ordered in ascending order according to the angle formed by their location with respect to the depot. This sorting method, based on angular positioning, facilitates the creation of efficient routes, as it ensures that customers closer to the depot are visited first. Subsequently, after sorting customers by angle, the Sweep algorithm allocates customers to vehicles while considering each vehicle's maximum capacity.

  - Density-based clustering: DBSCAN algorithm can be employed to generate clusters. We utilized a version similar to the one described in [**inproceedings**]. This version takes two parameters as input: *eps*, representing the radius value, and *min_points*, indicating the minimum number of points required for each cluster. The algorithm begins by designating a point as the center of each cluster. It then proceeds to examine the other points: for each cluster, it identifies the nearest neighbors of the selected point within a circle of radius *eps*. If this circle contains at least *min_points* points, the current point is considered part of the cluster. Once a cluster reaches its capacity, the process is restarted. This continues until all clusters meet their capacity constraints.
  - Random zones generation: it generates some rectangular zone in a similar way as the *zone_removal* function described in destroy/reconstruct chapter, then it adds point until the capacity constraints are met.
  - Random radial generation: similar to the last mechanism, but we consider circular zones, in a radial way.

- *Route generation in clusters*: this can happen in two ways: or totally random, or greedy (we take for every visited point the nearest as next point in the route).

## 4.2 Algorithm Examples

In this section we will show two examples of metaheuristic algorithms built using as building blocks the modules described in the last section. The first and the second algorithms are based on the VNS paradigm, with some differences from standard VNS. In this initial VNS framework (Algorithm 6), we incorporate all the characteristic features of VNS, with several additional enhancements. For instance, we integrate a hill-climbing mechanism to handle situations where there's no improvement in the solution. This mechanism operates with a probability derived from the Simulated Annealing algorithm. Furthermore, our shaking phase is slightly modified; it has a "boosted" aspect, wherein there's a probability, dependent on the number of iterations, for the shaking phase to transition into a destruction/reconstruction phase. The probability distribution utilized in this context is specifically designed to initiate with a probability of 0.5 at the outset, peak in the middle phase, and gradually decrease towards the end of the iterations. This ensures fewer destructions towards the end of the algorithm run.Furthermore, we introduce another mechanism: the incorporation of improved solutions into a taboo list (inspired by Taboo Search). From this list, two options are available: firstly, it can be utilized in conjunction with the crossover option, allowing for crossover operations if the taboo list exceeds a certain length defined by $len_taboo$ and the crossover option is set to true. The other use is the classical function taken from Taboo Search, if the improvement phase takes to a yet explored solution.

**Algorithm 6** Variable Neighborhood Search 1 (VNS1)

Routes *routes*, solution *sol*, points *points*, demands *demands*, capacity $Q$, iterations $T$, max improvement steps *hmax*, temperature *temperature*, length of taboo list *len_taboo*, improvement options *improvement*, cross-over option *cross_over*
Optimal routes *routes*, optimal solution *sol*

```
Initialize taboo list taboo as empty
t ← 0
tp ← temperature
while t < T do
    destruction_prob ← exp(−(5/2 × t/T − 0.8325)^2)
    (x1, sol1) ← Shake(sol, routes, points, demands, Q, destruction_prob)
    (x2, sol2) ← Improve(sol1, x1, points, demands, Q, hmax, improvement[1],
    improvement[0])
    feasible ← CheckConstraints(x2, demands, Q)
    if sol2 − sol < 0 and feasible then
        routes ← x2
        sol ← sol2
        Add (routes, sol) to taboo
        if length of taboo = len_taboo and t < T − 1 and cross_over then
            (routes, sol) ← CrossOver(taboo, points, demands, Q)
        end
    end
    else if sol2 − sol ≥ 0 and feasible then
        annealing_prob ← exp((sol − sol2) / tp)
        hill_climb ← Randomly choose 0 or 1 with probabilities (1−annealing_prob)
        and
        annealing_prob
        if hill_climb = 1 then
            routes ← x1
            sol ← sol1
        end
    end
    Increment t by 1
    Update temperature tp
end
if taboo is not empty then
    Find best solution in taboo based on sol values
    return Optimal routes and solution
end
else
    return routes and sol
end
```

**Algorithm 7** Variable Neighborhood Search 2 (VNS2)

**Input :** Routes *routes*, solution *sol*, points *points*, demands *demands*, capacity $Q$, number of iterations $T$, maximum number of improvement steps *hmax*,length of taboo list $len_t aboo$, improvement options *improvement*, cross-over option *cross_over*

**Output :** Optimal routes *routes*, optimal solution *sol*

```
Initialize taboo list taboo with empty list
t ← 0
while t < T do
    Apply destroy/repair operation to routes and sol
    Apply improvement operation to the obtained solution
    if new solution is better and feasible then
        Update routes and sol
        if taboo list is full and cross_over option is active then
            Apply cross-over using the taboo list
        end
        Add the new solution to the taboo list
        Increment t by 1
    else
        while new solution is not better do
            Apply another improvement operation
            if new solution is better and feasible then
                Update routes and sol
                if taboo list is full and cross_over option is active then
                    Apply cross-over using the taboo list
                end
                Add the new solution to the taboo list
                Increment t by 1
                break
            end
        end
    end
end
return Best solution from the taboo list
```

As you can observe, there's no shaking phase in this process; instead, solution perturbation occurs through a destroy/repair phase. Additionally, we don't apply the perturbation phase in every iteration due to its intensity. Instead, we continue applying improvements until further enhancement becomes impossible. At that point, we increment a counter and initiate the destruction/reconstruction phase.

Differently from the first algorithm, the incorporation of improved solutions into a taboo list (inspired by taboo search) is used only for crossing-over operations. This approach may appear similar to ILS, but it differs significantly. In ILS, the solution is completely reshaped during each restart, while in this case, we heavily perturb the current local solution while retaining some elements from the original. However, for ILS framework, we deployed something:

**Algorithm 8** Iterated Local Neighborhood Search (ILNS)

**Input:** points, demands, Q, T, hmax, len$_t$*aboo*, *improvement*, *cross*$_o$*ver*
**Output:** routes, sol

```
t ← 0;
sols ← [];
while t < T do
    Generate clusters around points and obtain initial routes;
    sql Copy code if length(sols) == len_taboo and cross_over then
        Apply cross-over using solutions in the taboo list
    end
    taboo ← []
    taboo_violated ← False
    Add initial routes to the taboo list
    while not taboo_violated do
        Apply improvements to the routes and obtain a new solution
        if the new solution is improved and feasible then
            Update the routes and the solution
            Add the new solution to the taboo list
            if length(taboo) == len_taboo and cross_over then
                Apply cross-over using solutions in the taboo list
            end
        end
        else if the new solution is not improved and feasible then
            Check if the solution is already in the taboo list
            If not, add it and terminate the loop
        end
    end
    Increment t
end
if there are valid solutions in the taboo list then
    Find the best solution in the taboo list;
    Return the routes and the best solution
end
```

This algorithm is quite similar to the one developed before, but the difference is that we not perturb our solution when it stops to improve, but we generate another one.

**Algorithm 9** Iterated Variable Neighborhood Search (IVNS)

**Input:** Routes *routes*, solution *sol*, points *points*, demands *demands*, capacity $Q$, number of iterations $T$, maximum number of improvement steps *hmax*, length of taboo list *len_taboo*, improvement options *improvement*, crossover option *cross_over*, shake flag *shak*

**Output:** Optimal routes *routes*, optimal solution *sol*

Initialize taboo list *taboo* as an empty list
Initialize iteration counter $t \leftarrow 0$
**while** $t < T$ **do**
    Calculate crossover probability $crossProb = \exp\left(-\left(\frac{5}{2}\frac{t}{T} - 0.8325546111576977\right)^2\right)$
    **if** *shak* **then**
        Perform shake operation: $x1, sol1, probs = shake(sol, routes, points, demands, Q,$
        "perturb"=0, "probs"=probs) **end**
        **else**
            $x1 \leftarrow routes$
            $sol1 \leftarrow sol$
        **end**
        Perform improvement operation: $x2, sol2 = improve(sol1, x1, points, demands, Q, hmax,$ "first" $= improvement[1],$ "mode" $= improvement[0])$
        Check feasibility of the new solution: $feasible, \_, \_ = constraints(x2, demands, Q)$
        **if** $sol2 - sol < 0$ ***and*** $feasible$ **then**
            Update routes and sol: $routes \leftarrow x2$, $sol \leftarrow sol2$
            Add the new solution to taboo list: *taboo*.append($(routes, sol)$)
            **if** *length of taboo list* == *len_taboo* ***and*** $t < T - 1$ ***and*** *cross_over* **then**
                Apply crossover using taboo list: $routes, sol \leftarrow$ mixing(taboo, points, demands, Q, crossProb)
            **end**
        **end**

```
else if sol2 − sol ≥ 0 and feasible then
    Check if the solution violates taboo
    if taboo is violated and t < T − 1 then
        Reset to a previous solution
        while feasibility is not achieved do
            Generate clusters: labels ← generate_clusters(points, demands, Q)
            Obtain initial routes: routes, sol ← first_route(points, labels,
            max(labels) + 1)
            Check feasibility: feasible, _, _ ← constraints(routes, demands, Q)
        end
        Continue to the next iteration
    end
end
Increment iteration counter: t ← t + 1
    end
    if taboo is not empty then
        Find the best solution in the taboo list
        Return the optimal routes and solution
    end
    else
        Return the current routes and solution
    end
```

The last algorithm that we will discuss is the *Iterated Variable Neighborhood Search*, which is a hybrid creature between VNS and ILS. Here we have some characteristics from both the "parent" algorithms: we have the shaking phase from VNS and a new starting routes generation phase as in ILS (which is activated when we have no more improvements). Also, in this algorithm, we have the chance to have a cross-over between solutions, but in this case, it can happen when we have an improvement of our solution (in order to perturb the solution in a beneficial manner).

# Chapter 5

# Experiments

In this chapter we will go to do some tests on our tools, in order to evaluate some aspects of their performances.

## 5.1 Examples

We can see four example solutions from the [**Uchoa_2017**] dataset of instances, in particular the A-n32-k5:

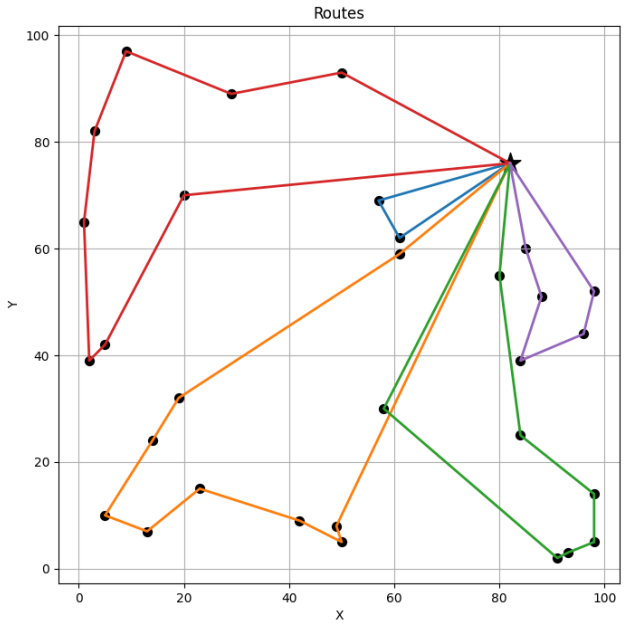


**Figure 5.1:** A-n32-k5 orTools

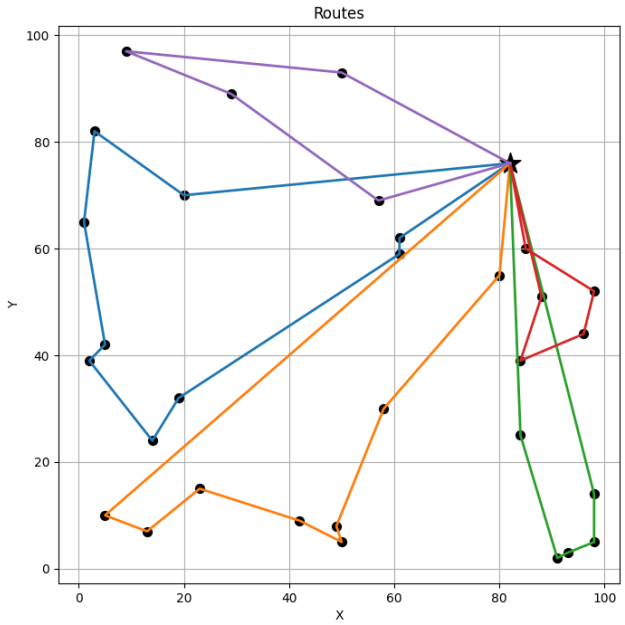


**Figure 5.2:** A-n32-k5 IVNS

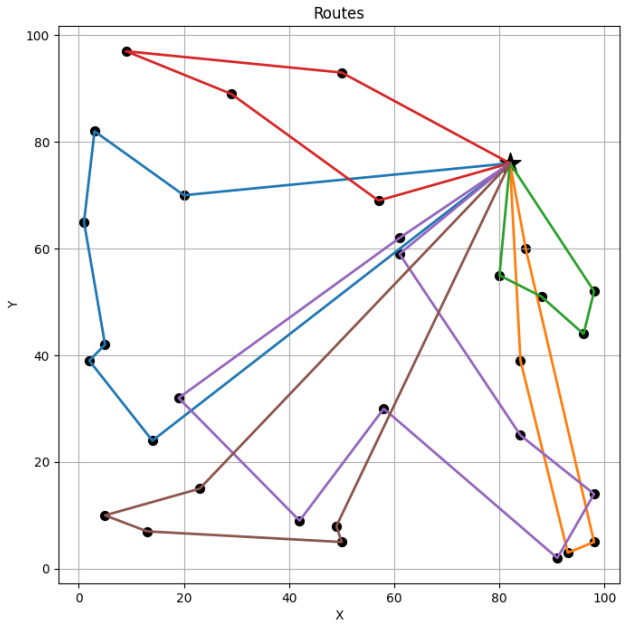


**Figure 5.3:** A-n32-k5 VNS1

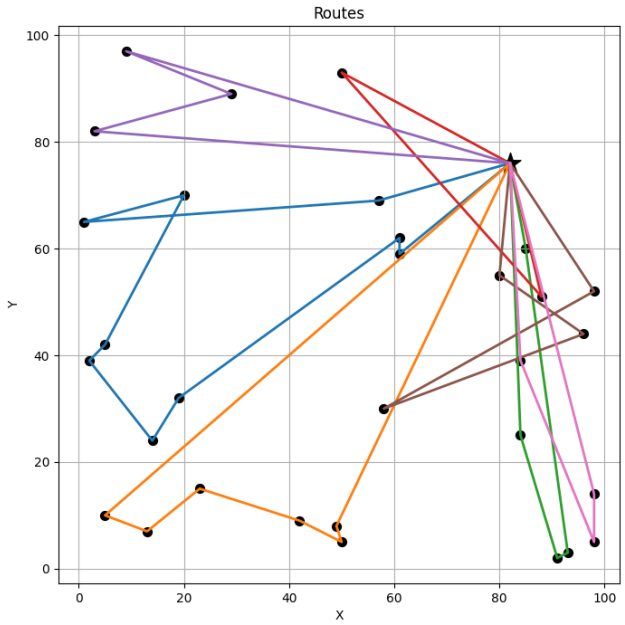


**Figure 5.4:** A-n32-k5 VNS2

## 5.2 Statistical Analysis: Non-parametric hypotesis Tests

In this part we will do a statistical analysis in order to test different aspects of our algorithms. In particular, as you'll going to see in the next section, we will take some *Non-parametric Statistical Tests* in order to compare our results.

In inferential statistics, hypothesis testing is used to draw conclusions about populations based on sample data. This involves defining two hypotheses: the null hypothesis ($H_0$), suggesting no effect or difference, and the alternative hypothesis ($H_1$), indicating the presence of an effect or difference (such as significant differences between algorithms). A significance level ($\alpha$) is chosen to determine when to reject the null hypothesis. Instead of predefining $\alpha$, the p-value can be computed, representing the probability of obtaining a result as extreme or more extreme than the observed result, assuming $H_0$ is true. As seen in [**Derrac_2011**], *nonparametric tests*, originally for nominal or ordinal data, can be adapted for continuous data using ranking-based transformations. They can conduct pairwise comparisons or multiple comparisons. Pairwise tests compare two algorithms at a time, while multiple comparisons tests compare more than two algorithms. In $1 \times N$ comparisons, a control method (the best-performing algorithm) is identified, and post-hoc procedures test for equality between the control method and the others. $N \times N$ comparisons consider equality hypotheses between all pairs of algorithms, with specific post-hoc procedures for this purpose.

In our analysis we are going to use two kinds of test in particular: the Wilcoxon Signed Rank test and the Friedman test.

### 5.2.1 Wilcoxon Signed Rank test

The Wilcoxon signed ranks test answers the following question: "do two samples represent two different populations? " [**Derrac_2011**] It is a statistical pairwise rank test used to compare the performance of two algorithms across a set of problems. Here's an elaboration of the test procedure:

1. **Calculating Differences:** For each problem in the dataset, compute the difference ($d_i$) between the performance scores of the two algorithms. If the scores are on different scales, consider normalizing them to the interval $[0, 1]$ to ensure fair comparison.
2. **Ranking Differences:** Rank the absolute values of the differences obtained in the previous step. In case of ties (i.e., when multiple differences have the same absolute value), various methods from the literature can be employed to handle ties. It's recommended to use average ranks for dealing with ties.

3. **Summing Ranks:** Calculate the sum of ranks for the positive differences ($R^+$) and the sum of ranks for the negative differences ($R^-$) as follow:

$$R^+ = \sum_{d_i>0} \text{rank}(d_i) + \frac{1}{2} \sum_{d_i=0} \text{rank}(d_i)$$

$$R^- = \sum_{d_i<0} \text{rank}(d_i) + \frac{1}{2} \sum_{d_i=0} \text{rank}(d_i)$$

   $R^+$ represents the sum of ranks for cases where the first algorithm outperformed the second, while $R^-$ represents the sum of ranks for cases where the second algorithm outperformed the first. Ranks of $d_i = 0$ are evenly distributed between $R^+$ and $R^-$, with any odd number of zero differences disregarded.

4. **Test Statistic:** Compute the test statistic ($T$) as $\min\{R^+, R^-\}$. If $T$ is less than or equal to the critical value from the *Wilcoxon distribution* for $n$ degrees of freedom, the null hypothesis of equality of means is rejected. This indicates that one algorithm significantly outperforms the other, with the associated $p$-value providing further statistical significance.

The Wilcoxon signed ranks test is more sensitive than the t-test and does not assume normal distributions, making it a safer option for comparing algorithms. It also handles outliers more effectively. It's important to ensure that the differences ($d_i$) are continuous and not rounded, as rounding may decrease the test's power, especially in the case of ties.

### 5.2.2 Friedman test

The Friedman test is a multiple test, before talking about it, we have to make few consideration. n pairwise analysis, attempting to draw conclusions involving multiple pairwise comparisons results in an accumulated error stemming from their combination. In statistical terms, this means losing control over the Family-Wise Error Rate (FWER), which is defined as the probability of making one or more false discoveries among all the hypotheses when conducting multiple pairwise tests:

$$\begin{aligned} p &= P(\text{Reject } H_0 | H_0 \text{ true}) = 1 - P(\text{Accept } H_0 | H_0 \text{ true}) \\ &= 1 - P(\text{Accept } A_k = A_i, i = 1, \ldots, k-1 | H_0 \text{ true}) \\ &= 1 - \prod_{i=1}^{k-1} P(\text{Accept } A_k = A_i | H_0 \text{ true}) \\ &= 1 - \prod_{i=1}^{k-1} [1 - P(\text{Reject } A_k = A_i | H_0 \text{ true})] \end{aligned}$$

$$= 1 - \prod_{i=1}^{k-1} (1 - p_{Hi})$$

This value computed above is the probability of reject the Null Hypotesis, given that it's true(first type error). As you can see, for multiple test, we can get an high probability(values close to 1) to make a type I error. Therefore, employing a pairwise comparison test, such as Wilcoxon's test, to perform multiple comparisons across a set of algorithms is not advisable, as it does not control the FWER. In order to face this issue, we can use $1 \times N$ comparisons.Here, a control method is defined as one algorithm of primary interest, as the best performing algorithm . The *Friedman test* is used to answer the following question ([**Derrac_2011**]): "in a set of $k$ samples (where $k \geq 2$), do at least two of the samples represent populations with different median values?".

The Friedman test serves as the nonparametric equivalent of the repeated measures ANOVA, enabling the detection of significant differences among the performance of two or more algorithms. The null hypothesis of Friedman's test posits equality of medians across the populations, while the alternative hypothesis is non-directional.

To calculate the test statistic, the original results are first converted into ranks following these steps:

1. Gather observed results for each algorithm/problem pair.
2. Rank values from 1 (best result) to $k$ (worst result) for each problem $i$. Denote these ranks as $r_{ji}$, where $1 \leq j \leq k$.
3. Average the ranks obtained for each algorithm $j$ across all problems to obtain the final rank $R_j = \frac{1}{n} \sum_i r_{ji}$. In case of ties, computing average ranks is recommended.

Under the null hypothesis, where all algorithms behave similarly and their ranks $R_j$ should be equal, the Friedman statistic $F_f$ can be computed as:

$$F_f = \frac{12n}{k(k+1)} \left[ \sum_j R_j^2 - \frac{k(k+1)^2}{4} \right]$$

This statistic follows a chi-squared distribution with $k - 1$ degrees of freedom when both $n$ and $k$ are sufficiently large (typically $n > 10$ and $k > 5$).

The main drawback of the Friedman test is its inability to establish proper comparisons between some of the algorithms considered, as it can only detect significant differences over the entire multiple comparison. When the goal of applying multiple tests is to compare a control method with a set of algorithms, a family of hypotheses can be defined, all related to the control method. Then, the application of a post-hoc test can provide a p-value determining the degree of rejection of each hypothesis.

A family of hypotheses consists of logically interrelated comparisons, wherein 1×N comparisons compare the $k-1$ algorithms of the study (excluding the control) with the control method, and N×N comparisons consider the $\frac{k(k-1)}{2}$ possible comparisons among algorithms. Consequently, the family comprises $k-1$ or $\frac{k(k-1)}{2}$ hypotheses, respectively, ordered by their p-values from lowest to highest.

The p-value of each hypothesis in the family can be obtained by converting the rankings computed by each test using a normal approximation. The test statistic for comparing the $i$th algorithm and $j$th algorithm, denoted by $z$, depends on the primary nonparametric procedure used. In this case, using Friedman Statistic:

$$z = \frac{(R_i - R_j)}{\sqrt{\frac{k(k+1)}{6n}}}$$

When a p-value is considered in a multiple test, it reflects the probability error of a certain comparison, but it does not take into account the remaining comparisons belonging to the family. If k algorithms are being compared and in each comparison the level of significance is $\alpha$, then in a single comparison the probability of not making a Type I error (rejecting a true null hypothesis) is $(1\alpha)$, and the probability of not making a Type I error in the k  1 comparison is $(1\alpha)(k1)$. Therefore, the probability of making one or more Type I error is $1(1\alpha)^{(k1)}$.

The z-value in all cases is used to find the corresponding probability (p-value) from the table of normal distribution $N(0,1)$, which is then compared with an appropriate level of significance $\alpha$ . The post-hoc tests differ in the way they adjust the value of $\alpha$ to compensate for multiple comparisons.

Between the possible p-value corrections, we chose the *Holm Post-hoc procedure*. The Holm Post-hoc procedure uses a step-down approach to apply the corrections: let $p_1, p_2, \ldots, p_{k-1}$ be the ordered p-values (smallest to largest), so that $p_1 \leq p_2 \leq \cdots \leq p_{k-1}$, and let $H_1, H_2, \ldots, H_{k-1}$ be the corresponding hypotheses. The Holm procedure rejects $H_1$ to $H_{i-1}$ if $i$ is the smallest integer such that $p_i > \alpha/(k-1)$. Holm's step-down procedure starts with the most significant p-value. If $p_1$ is below $\alpha/(k-1)$, the corresponding hypothesis is rejected and we are allowed to compare $p_2$ with $\alpha/(k-2)$. If the second hypothesis is rejected, the test proceeds with the third, and so on. As soon as a certain null hypothesis cannot be rejected, all the remaining hypotheses are retained as well.

## 5.3 Experiments results

In order to do some tests on our algorithms, we have to different instances of the CVRP problem. We used instances taken from [**Uchoa_2017**]: the instances described here comes from different sets generated in the years. We used instances from these sets: set A, set B, set E, set P and set CMT (we have about 103

instances to be tested). Before talking about experiments, we have to say that our experiments are taken using an Acer Nitro 5 with an AMD Ryzen 7 5800H (3.20 GHz), and 16 GB.

### 5.3.1 Experiment 1

A first experiment has been taken running the OrTools solver in automatic mode against three of our algorithms; in particular, we used VNS1,VNS2 and IVNS described in the Chapter 3. In this case, we used these sets of parameters for every algorithm:

- VNS1: `T`: 10, `hmax`: 10, `temperature`: 20, `len_taboo`: 10, `improvement`: ('3bis', False),`cross_over`: False
- VNS2: `T`: 5, `hmax`: 10, `temperature`: 20, `len_taboo`: 10, `improvement`: ('3bis', False),`cross_over`: False
- IVNS: `T`: 10, `hmax`: 10, `temperature`: 20, `len_taboo`: 10, `improvement`: ('3bis', False),`cross_over`: False

The parameter `improvement` given as the tuple "('3bis', False)" means that we are using an improvement algorithm with diversification and intensification phase with best improvement choice cryteria. The starting solution of all our algorithms has been found using sweep algorithm initialization.

In this test, firstly we take 10 repetitions for every tested instance, because for different tests our algorithms can give slightly different results.

**Table of average test results for every instance**

| InstanceName | orTools | VNS1 | VNS2 | IVNS |
|---|---|---|---|---|
| A-n62-k8 | 1491 | 1794.496535 | 2257.418974 | 1648.606403 |
| A-n34-k5 | 842 | 896.8505891 | 919.791782 | 822.4983152 |
| A-n45-k6 | 1049 | 1199.10867 | 1293.805111 | 1026.319145 |
| A-n33-k5 | 761 | 787.6574833 | 851.1978253 | 721.3509776 |
| A-n65-k9 | 1258 | 1457.184303 | 1657.759698 | 1336.123692 |
| A-n33-k6 | 797 | 875.3616018 | 908.9787625 | 800.4170787 |
| A-n39-k6 | 883 | 1073.413187 | 1207.978065 | 913.4509548 |
| A-n69-k9 | 1203 | 1477.017694 | 1673.84738 | 1310.267719 |
| A-n36-k5 | 853 | 958.4813399 | 1143.379111 | 888.7114965 |
| A-n37-k5 | 721 | 854.5904692 | 939.1207003 | 772.8580969 |
| A-n63-k10 | 1400 | 1720.428882 | 2029.44528 | 1601.839782 |
| A-n46-k7 | 1038 | 1186.450568 | 1353.436445 | 1061.236829 |
| A-n54-k7 | 1226 | 1604.787655 | 1889.367299 | 1401.241494 |
| A-n45-k7 | 1148 | 1416.083014 | 1608.013015 | 1316.061848 |
| A-n63-k9 | 1799 | 2305.534787 | 2596.662257 | 1976.774502 |
| A-n48-k7 | 1145 | 1310.993737 | 1615.338963 | 1210.981358 |
| A-n37-k6 | 1011 | 1221.513758 | 1379.753226 | 1048.92581 |
| A-n64-k9 | 1477 | 2029.37495 | 2387.516528 | 1835.219801 |
| A-n39-k5 | 861 | 999.8565589 | 1115.773839 | 904.8154337 |
| A-n80-k10 | 1915 | 2983.456708 | 3203.043554 | 2398.375354 |
| A-n44-k6 | 995 | 1260.792864 | 1401.14992 | 1055.692839 |
| A-n53-k7 | 1098 | 1340.501411 | 1463.334932 | 1129.694789 |
| A-n32-k5 | 782 | 1087.17936 | 1243.628311 | 876.1122758 |
| A-n61-k9 | 1189 | 1327.039948 | 1386.425484 | 1177.744934 |
| A-n60-k9 | 1494 | 1852.992369 | 2097.529975 | 1636.675245 |
| A-n38-k5 | 821 | 993.7565437 | 1119.345715 | 842.8556829 |
| A-n55-k9 | 1148 | 1408.933633 | 1535.745506 | 1236.763166 |

As seen in the last section, we are going to do some nonparametric tests on our data, in order to know something about our algorithms performances. In this case, we have to compare 4 algorithms, so that we have to use an $1 \times N$ comparison, following this path as shown in [**Derrac_2011**]. Before computing our tests, we normalize our datas, in order scale problem instances differences and we can focus only on the performance of the algorithms relative to each other. The normalization for every simulation taken is calculated in this way:

$$\tilde{x}_i = \frac{x_i - x_{\texttt{min}}}{x_{\texttt{max}} - x_{\texttt{min}}}$$

Where $\tilde{x}_i$ is normalized value and it is in [0,1] ; $x_i$, $x_{\texttt{min}}$ and $x_{\texttt{max}}$ are respectively the value of our simulation for algorithm i, minimum value in the simulation and maximum value in the simulation. Same normalization technique has been used for computation times.

First of all, we compute average ranks of our algorithms.

After that we compute the value of the Friedman Statistic and calculate it's value of CDF chi-squared distribution with $k = 4$ degrees of freedom(we are testing 4 algorithms): if our Friedman statistics values allow to have values of $1 - \texttt{cdf}(F)$ greater then our Family-wise error rate ($\alpha = 0.1$ in this case, we can claim that the samples have different median value with that significance(rejecting the null hypotesis), so that it has sense proceeding the paired tests and post-hoc procedures. In this experiments we got:

$$F_{solutions} = 2569.2288349514565$$
$$F_{times} = 2875.6310679611647$$

Both the values are showing us that there are difference between populations, so we can continue our tests.

In this case we will apply paired tests computing p-values using firstly a Z-statistic ad-hoc computed for Friedman ranks, as explained in the Friedman test description. Then, when we will get the p-values, we can apply our Holm-Bonferroni corrections. We apply these procedures with 3 different alpha levels ($\alpha = 0.1, 0.05, 0.01$.

As it's possible to see, we can see that the most powerful algorithm is the oR-tools framework, as predicted. Focusing our analysis on our algorithms, we can see that in terms of solution minimization the best performing algorithm is IVNS while the worst performing is VNS2 in the pairwise tests. In the context of solution time, the best performing algorithm is newly IVNS and the worst times in this case are from VNS2. VNS2 is the worst because it needs more iterations to complete the search, so in this case it doesn't work very well.

Last but not least, we make a pairwise test using a non parametric test, the Wilcoxon test, and applying on it Holm corrections, as in [**Souza_2023**]. These are the results:

**Table of average test results for every instance**

| InstanceName | orTools | VNS1 | VNS2 | IVNS |
|---|---|---|---|---|
| B-n63-k10 | 1600 | 1943.670482 | 2005.913758 | 1811.319539 |
| B-n43-k6 | 755 | 895.8178567 | 927.8966126 | 848.555234 |
| B-n52-k7 | 742 | 901.6224899 | 911.2131056 | 855.6070509 |
| B-n57-k7 | 1137 | 1366.350248 | 1421.856297 | 1246.043839 |
| B-n45-k5 | 790 | 980.2636894 | 1082.467977 | 893.5521465 |
| B-n56-k7 | 756 | 895.1043214 | 975.9191382 | 803.1514304 |
| B-n78-k10 | 1272 | 1507.991575 | 1683.00878 | 1453.616824 |
| B-n35-k5 | 983 | 1087.495266 | 1118.696392 | 1046.612076 |
| B-n67-k10 | 1084 | 1256.816506 | 1322.47078 | 1186.844017 |
| B-n41-k6 | 854 | 989.1025853 | 1021.99331 | 980.1486111 |
| B-n45-k6 | 700 | 819.7430413 | 877.6129911 | 772.9240391 |
| B-n64-k9 | 1078 | 1009.343417 | 1018.548062 | 970.7416247 |
| B-n38-k6 | 866 | 885.7753401 | 926.6882709 | 834.4984453 |
| B-n68-k9 | 1356 | 1625.206719 | 1824.458137 | 1487.946598 |
| B-n44-k7 | 929 | 1126.250202 | 1265.280099 | 1064.261603 |
| B-n51-k7 | 1135 | 1281.142278 | 1291.928288 | 1217.336839 |
| B-n57-k9 | 1662 | 1913.911888 | 2100.672926 | 1770.15092 |
| B-n34-k5 | 811 | 952.8073316 | 976.0872646 | 913.9035379 |
| B-n39-k5 | 601 | 790.5846646 | 874.9953482 | 706.653451 |
| B-n66-k9 | 1366 | 1479.708775 | 1569.073879 | 1428.003351 |
| B-n50-k7 | 742 | 921.5675221 | 998.1638576 | 847.9386545 |
| B-n31-k5 | 668 | 839.4331673 | 926.0874982 | 733.7239465 |
| B-n50-k8 | 1311 | 1510.209409 | 1734.368311 | 1406.714146 |

**Table of average test results for every instance**

| InstanceName | orTools | VNS1 | VNS2 | IVNS |
|---|---|---|---|---|
| E-n76-k7 | 718 | 1047.731654 | 1144.012837 | 861.6294189 |
| E-n76-k14 | 1189 | 1297.484674 | 1504.1321 | 1222.714474 |
| E-n33-k4 | 879 | 1088.662372 | 1235.38634 | 935.738517 |
| E-n76-k10 | 885 | 1099.189183 | 1258.072128 | 987.0985989 |
| E-n22-k4 | 425 | 425.1172306 | 429.121522 | 400.494638 |
| E-n76-k8 | 754 | 1019.221074 | 1200.157286 | 909.4423236 |
| E-n23-k3 | 559 | 599.7139542 | 622.5857964 | 594.5187361 |
| E-n101-k8 | 845 | 1128.40402 | 1335.785432 | 1062.214661 |
| E-n30-k3 | 531 | 653.7806189 | 739.8806448 | 569.260037 |
| E-n51-k5 | 568 | 683.5612911 | 789.0553832 | 617.21473 |
| E-n101-k14 | 1133 | 1433.049969 | 1657.225029 | 1390.768086 |

**Table of average test results for every instance**

| InstanceName | orTools | VNS1 | VNS2 | IVNS |
|---|---|---|---|---|
| P-n20-k2 | 222 | 299.2244899 | 323.6047567 | 256.7594554 |
| P-n70-k10 | 898 | 1071.055068 | 1167.475376 | 935.2526665 |
| P-n60-k15 | 1003 | 1193.773599 | 1304.394934 | 1113.131659 |
| P-n76-k5 | 687 | 916.9800643 | 1040.243839 | 821.2898889 |
| P-n19-k2 | 222 | 273.4000364 | 299.9846594 | 250.4396755 |
| P-n55-k10 | 720 | 896.9569559 | 984.1906514 | 769.5886745 |
| P-n51-k10 | 818 | 922.3986252 | 1022.435056 | 860.2461536 |
| P-n22-k8 | 583 | 671.2770837 | 680.0616196 | 671.0750963 |
| P-n22-k2 | 212 | 274.6150297 | 310.571317 | 221.4577595 |
| P-n101-k4 | 767 | 1007.875302 | 1223.450865 | 971.6600368 |
| P-n50-k8 | 665 | 779.1390904 | 913.7297157 | 697.983893 |
| P-n21-k2 | 208 | 261.5080323 | 306.0866455 | 216.819447 |
| P-n40-k5 | 449 | 575.8571862 | 642.4623107 | 515.913487 |
| P-n60-k10 | 824 | 993.7486808 | 1127.369044 | 856.889331 |
| P-n55-k15 | 968 | 1143.865526 | 1215.01845 | 1080.842224 |
| P-n23-k8 | 530 | 626.1916859 | 657.5600357 | 616.7067482 |
| P-n65-k10 | 826 | 958.7263869 | 1099.541634 | 865.2634822 |
| P-n50-k7 | 559 | 717.9526211 | 798.0464466 | 616.6728528 |
| P-n76-k4 | 655 | 781.347489 | 1012.922829 | 795.1705004 |
| P-n45-k5 | 533 | 675.4294065 | 719.6409244 | 570.2460832 |
| P-n55-k7 | 561 | 753.9105076 | 865.4309413 | 645.4964257 |
| P-n50-k10 | 759 | 890.4709556 | 965.7660899 | 789.2854755 |
| P-n55-k8 | 607 | 743.4627428 | 880.6017014 | 663.3723822 |

**Table of average test results for every instance**

| **InstanceName** | **orTools** | **VNS1** | **VNS2** | **IVNS** |
|---|---|---|---|---|
| CMT5 | 1411 | 2342.936411 | 2563.748521 | 2100.23106 |
| CMT4 | 1092 | 1722.928981 | 2104.408847 | 1591.736948 |
| CMT2 | 885 | 1121.656439 | 1221.989646 | 981.1835279 |
| CMT13 | 1058 | 2063.005221 | 3064.108193 | 2090.920389 |
| CMT11 | 1058 | 2262.829871 | 2993.831252 | 2101.669984 |
| CMT6 | 568 | 692.3888339 | 790.4232874 | 616.1034828 |
| CMT1 | 568 | 690.0795287 | 797.9156114 | 618.7928406 |
| CMT8 | 845 | 1152.072426 | 1326.311771 | 1062.649433 |
| CMT7 | 885 | 1084.712328 | 1238.107714 | 982.2678601 |
| CMT9 | 1092 | 1547.948444 | 2126.530483 | 1594.885697 |
| CMT3 | 845 | 1145.975595 | 1331.722311 | 1061.789806 |
| CMT14 | 910 | 1106.115789 | 1174.649188 | 1061.974693 |
| CMT12 | 910 | 1136.381238 | 1176.025438 | 1060.894122 |
| CMT10 | 1411 | 2051.996009 | 2622.351323 | 2116.446969 |

**Table of average test time executions for every instance(in seconds)**

| InstanceName | orTools | VNS1 | VNS2 | IVNS |
|---|---|---|---|---|
| A-n62-k8 | 0.26840564 | 3.51545959 | 1.50604888 | 3.01736504 |
| A-n34-k5 | 0.07081066 | 1.70438014 | 0.7950107 | 1.62472454 |
| A-n45-k6 | 0.15735312 | 2.29730819 | 1.05431362 | 2.00181949 |
| A-n33-k5 | 0.06781842 | 1.58126767 | 0.74064297 | 1.55741365 |
| A-n65-k9 | 0.30588381 | 3.37368755 | 1.60482402 | 2.92115646 |
| A-n33-k6 | 0.06356641 | 1.48608601 | 0.70838998 | 1.42417538 |
| A-n39-k6 | 0.1008356 | 2.00086966 | 0.89482581 | 1.8884321 |
| A-n69-k9 | 0.34141649 | 3.65500008 | 1.78689146 | 3.15479044 |
| A-n36-k5 | 0.09738043 | 2.07722266 | 0.89652523 | 1.98843448 |
| A-n37-k5 | 0.11048875 | 2.06454754 | 0.93932859 | 2.10334845 |
| A-n63-k10 | 0.29858019 | 3.35880588 | 1.57732476 | 2.67214738 |
| A-n46-k7 | 0.14107238 | 2.37391127 | 1.0170197 | 1.98733511 |
| A-n54-k7 | 0.27455175 | 2.87664079 | 1.34698389 | 2.53720145 |
| A-n45-k7 | 0.15843117 | 2.25341208 | 1.11971902 | 2.04548936 |
| A-n63-k9 | 0.25936826 | 3.08964434 | 1.48843133 | 2.70054527 |
| A-n48-k7 | 0.1488731 | 2.56392764 | 1.16085273 | 2.26825916 |
| A-n37-k6 | 0.14922079 | 1.82874614 | 0.8064266 | 1.55672201 |
| A-n64-k9 | 0.28065541 | 3.33825575 | 1.55433571 | 2.84359939 |
| A-n39-k5 | 0.08977566 | 2.0617489 | 0.9693572 | 1.94778319 |
| A-n80-k10 | 0.50144553 | 4.41496912 | 2.08269095 | 3.82149493 |
| A-n44-k6 | 0.14335713 | 2.27574103 | 1.06349887 | 2.07734182 |
| A-n53-k7 | 0.25156392 | 2.98115506 | 1.31951837 | 2.58297301 |
| A-n32-k5 | 0.08453157 | 1.51780257 | 0.70015807 | 1.44074119 |
| A-n61-k9 | 0.29033682 | 3.1429132 | 1.35848338 | 2.59401691 |
| A-n60-k9 | 0.23933668 | 3.03931359 | 1.47055297 | 2.60382401 |
| A-n38-k5 | 0.09337327 | 1.91123508 | 0.87592272 | 1.83101126 |
| A-n55-k9 | 0.20630359 | 2.74343634 | 1.27432065 | 2.37746044 |

**Table of average test time executions for every instance(in seconds)**

| InstanceName | orTools | VNS1 | VNS2 | IVNS |
|---|---|---|---|---|
| B-n63-k10 | 0.41643657 | 3.06020015 | 1.53091081 | 2.80261992 |
| B-n43-k6 | 0.25518623 | 2.29109687 | 1.0896064 | 2.07904145 |
| B-n52-k7 | 0.27415766 | 2.78000951 | 1.32780829 | 2.55218597 |
| B-n57-k7 | 0.24267793 | 3.00072826 | 1.49328092 | 2.71573035 |
| B-n45-k5 | 0.19940461 | 2.48250867 | 1.15639371 | 2.29358499 |
| B-n56-k7 | 0.1842562 | 3.10922224 | 1.4645811 | 2.86941175 |
| B-n78-k10 | 0.46081389 | 4.72219491 | 2.01474536 | 3.8303909 |
| B-n35-k5 | 0.08934885 | 1.76469594 | 0.77239644 | 1.60138158 |
| B-n67-k10 | 0.40561551 | 3.94673902 | 1.72888402 | 3.10945669 |
| B-n41-k6 | 0.11125838 | 1.94017191 | 0.90250727 | 1.94012081 |
| B-n45-k6 | 0.15653813 | 2.26967153 | 1.06636392 | 2.01956849 |
| B-n64-k9 | 0.39634036 | 3.4638299 | 1.59938003 | 2.91736419 |
| B-n38-k6 | 0.08203495 | 1.83116756 | 0.8114557 | 1.64685678 |
| B-n68-k9 | 0.38354748 | 3.88777197 | 1.69077528 | 3.18053441 |
| B-n44-k7 | 0.17868628 | 2.26153348 | 0.98808908 | 1.9505202 |
| B-n51-k7 | 0.14691705 | 2.64787915 | 1.21408579 | 2.29174932 |
| B-n57-k9 | 0.22877392 | 2.9091741 | 1.36129028 | 2.51372905 |
| B-n34-k5 | 0.06105837 | 1.62568887 | 0.79620012 | 1.55854589 |
| B-n39-k5 | 0.0908738 | 2.05535062 | 0.92758102 | 1.8870465 |
| B-n66-k9 | 0.30860868 | 3.49151959 | 1.67703034 | 3.09808092 |
| B-n50-k7 | 0.21088095 | 2.67066696 | 1.2221663 | 2.48586042 |
| B-n31-k5 | 0.05451231 | 1.41363905 | 0.69040514 | 1.33136096 |
| B-n50-k8 | 0.28685657 | 2.4799803 | 1.21031135 | 2.20610812 |

**Table of average test time executions for every instance(in seconds)**

| InstanceName | orTools | VNS1 | VNS2 | IVNS |
|---|---|---|---|---|
| E-n76-k7 | 0.77619076 | 11.65920753 | 5.28494582 | 11.25377392 |
| E-n76-k14 | 0.76010841 | 10.2827449 | 5.06423803 | 8.84505755 |
| E-n33-k4 | 0.20377105 | 4.1250141 | 1.88766106 | 4.01923816 |
| E-n76-k10 | 1.4142915 | 10.9255346 | 5.06332843 | 9.29147078 |
| E-n22-k4 | 0.03688593 | 0.95000096 | 0.45227441 | 0.97210596 |
| E-n76-k8 | 1.14599668 | 11.45349897 | 4.92534279 | 9.77355079 |
| E-n23-k3 | 0.03420376 | 1.52894843 | 0.70163404 | 1.70450183 |
| E-n101-k8 | 0.90520719 | 7.30397385 | 3.26506915 | 6.5271502 |
| E-n30-k3 | 0.16106929 | 4.60246055 | 1.87582241 | 4.71551929 |
| E-n51-k5 | 0.4881226 | 6.63708344 | 3.06676499 | 6.372933 |
| E-n101-k14 | 0.77183944 | 5.81119022 | 2.69264786 | 4.71761738 |

**Table of average test time executions for every instance(in seconds)**

| InstanceName | orTools | VNS1 | VNS2 | IVNS |
|---|---|---|---|---|
| P-n20-k2 | 0.02488078 | 1.07033461 | 0.49894967 | 1.35036184 |
| P-n70-k10 | 0.34479056 | 3.72023209 | 1.60083583 | 3.2233343 |
| P-n60-k15 | 0.22534516 | 2.75271281 | 1.33752462 | 2.3472114 |
| P-n76-k5 | 0.50697037 | 6.36749965 | 2.55556511 | 5.87925194 |
| P-n19-k2 | 0.03702522 | 1.04444499 | 0.46675049 | 1.23554723 |
| P-n55-k10 | 0.1658497 | 2.51877914 | 1.20507026 | 2.28326265 |
| P-n51-k10 | 0.22051633 | 2.22307333 | 1.06416599 | 2.1576255 |
| P-n22-k8 | 0.03271257 | 0.91594081 | 0.48101543 | 0.9037541 |
| P-n22-k2 | 0.0329244 | 1.4216605 | 0.65804403 | 1.85839263 |
| P-n101-k4 | 0.69425242 | 14.01204244 | 5.29263503 | 14.89160901 |
| P-n50-k8 | 0.16426155 | 2.25174001 | 1.06698225 | 2.01495598 |
| P-n21-k2 | 0.02943732 | 1.33580618 | 0.5810703 | 1.63833387 |
| P-n40-k5 | 0.12350357 | 2.07730538 | 0.97300477 | 1.996304 |
| P-n60-k10 | 0.22417144 | 2.81579797 | 1.36450849 | 2.54173794 |
| P-n55-k15 | 0.19173995 | 2.40955684 | 1.24385882 | 2.22301919 |
| P-n23-k8 | 0.04116628 | 0.94897762 | 0.45982233 | 0.88232223 |
| P-n65-k10 | 0.46247905 | 3.29518461 | 1.53463971 | 2.86074769 |
| P-n50-k7 | 0.20436216 | 2.39004053 | 1.12071202 | 2.28067492 |
| P-n76-k4 | 0.43053157 | 8.67402202 | 3.22263106 | 7.69715938 |
| P-n45-k5 | 0.12443104 | 2.43411175 | 1.11896796 | 2.5978495 |
| P-n55-k7 | 0.21034111 | 2.91747279 | 1.34388875 | 2.66687075 |
| P-n50-k10 | 0.20005682 | 2.29805713 | 1.11054709 | 2.05985701 |
| P-n55-k8 | 0.18299914 | 2.89116506 | 1.3022401 | 2.66912642 |

**Table of average test time executions for every instance(in seconds)**

| **InstanceName** | **orTools** | **VNS1** | **VNS2** | **IVNS** |
|---|---|---|---|---|
| CMT5 | 3.00040086 | 19.98776783 | 8.36640486 | 42.06073544 |
| CMT4 | 1.8104656 | 13.66240313 | 6.01880438 | 11.14295624 |
| CMT2 | 0.57166575 | 4.21609132 | 1.99684959 | 3.65293351 |
| CMT13 | 1.04505769 | 11.87911777 | 4.70295699 | 10.00895643 |
| CMT11 | 1.03828959 | 10.81802167 | 4.76041042 | 9.80813436 |
| CMT6 | 0.51524196 | 8.02347172 | 3.69875247 | 7.51636043 |
| CMT1 | 0.19875095 | 2.72158285 | 1.27453668 | 2.70472969 |
| CMT8 | 2.37855773 | 23.51340177 | 10.06839839 | 20.49450077 |
| CMT7 | 1.44867598 | 11.33820257 | 5.57696796 | 9.42337568 |
| CMT9 | 4.60320061 | 39.41257391 | 16.37902542 | 30.83443658 |
| CMT3 | 0.95168337 | 8.19664093 | 3.45380852 | 7.2479982 |
| CMT14 | 0.69723368 | 7.21823555 | 2.99935201 | 5.97802245 |
| CMT12 | 0.69154124 | 6.75093215 | 2.98893864 | 5.91968441 |
| CMT10 | 2.99937465 | 21.74737476 | 8.45042884 | 15.58288148 |

Average ranks for solution results values

| **InstanceName** | **orTools** | **VNS1** | **VNS2** | **IVNS** |
|---|---|---|---|---|
| **Average** Ranks | 1.11456311 | 2.92135922 | 3.87524272 | 2.08883495 |

Average ranks for computation times values

| **InstanceName** | **orTools** | **VNS1** | **VNS2** | **IVNS** |
|---|---|---|---|---|
| **Average Times** | 1. | 3.77669903 | 2. | 3.22330097 |

**Z-test on solutions' values**

| **Test** | **alpha = 0.1** | **alpha = 0.05** | **alpha = 0.01** |
|---|---|---|---|
| oR-tools < VNS1 | TRUE | TRUE | TRUE |
| oR-tools < VNS2 | TRUE | TRUE | TRUE |
| oR-tools < IVNS | TRUE | TRUE | TRUE |
| VNS1 < VNS2 | TRUE | TRUE | TRUE |
| VNS1 < IVNS | FALSE | FALSE | FALSE |
| VNS2 < IVNS | FALSE | FALSE | FALSE |
| oR-tools > VNS1 | FALSE | FALSE | FALSE |
| oR-tools > VNS2 | FALSE | FALSE | FALSE |
| oR-tools > IVNS | FALSE | FALSE | FALSE |
| VNS1 > VNS2 | FALSE | FALSE | FALSE |
| VNS1 > IVNS | TRUE | TRUE | TRUE |
| VNS2 > IVNS | TRUE | TRUE | TRUE |

**Z-test on computing times**

| **Test** | **alpha = 0.1** | **alpha = 0.05** | **alpha = 0.01** |
|---|---|---|---|
| oR-tools < VNS1 | TRUE | TRUE | TRUE |
| oR-tools < VNS2 | TRUE | TRUE | TRUE |
| oR-tools < IVNS | TRUE | TRUE | TRUE |
| VNS1 < VNS2 | FALSE | FALSE | FALSE |
| VNS1 < IVNS | FALSE | FALSE | FALSE |
| VNS2 < IVNS | TRUE | TRUE | TRUE |
| oR-tools > VNS1 | FALSE | FALSE | FALSE |
| oR-tools > VNS2 | FALSE | FALSE | FALSE |
| oR-tools > IVNS | FALSE | FALSE | FALSE |
| VNS1 > VNS2 | TRUE | TRUE | TRUE |
| VNS1 > IVNS | TRUE | TRUE | TRUE |
| VNS2 > IVNS | FALSE | FALSE | FALSE |

As it's possible to see, Wilcoxon rank tests confirm the results seen in Z-tests.

**Wilcoxon rank test on solutions' values**

| **Test** | **alpha = 0.1** | **alpha = 0.05** | **alpha = 0.01** |
|---|---|---|---|
| oR-tools < VNS1 | TRUE | TRUE | TRUE |
| oR-tools < VNS2 | TRUE | TRUE | TRUE |
| oR-tools < IVNS | TRUE | TRUE | TRUE |
| VNS1 < VNS2 | TRUE | TRUE | TRUE |
| VNS1 < IVNS | FALSE | FALSE | FALSE |
| VNS2 < IVNS | FALSE | FALSE | FALSE |
| oR-tools > VNS1 | FALSE | FALSE | FALSE |
| oR-tools > VNS2 | FALSE | FALSE | FALSE |
| oR-tools > IVNS | FALSE | FALSE | FALSE |
| VNS1 > VNS2 | FALSE | FALSE | FALSE |
| VNS1 > IVNS | TRUE | TRUE | TRUE |
| VNS2 > IVNS | TRUE | TRUE | TRUE |

**Wilcoxon rank test on computing times**

| **Test** | **alpha = 0.1** | **alpha = 0.05** | **alpha = 0.01** |
|---|---|---|---|
| oR-tools < VNS1 | TRUE | TRUE | TRUE |
| oR-tools < VNS2 | TRUE | TRUE | TRUE |
| oR-tools < IVNS | TRUE | TRUE | TRUE |
| VNS1 < VNS2 | FALSE | FALSE | FALSE |
| VNS1 < IVNS | FALSE | FALSE | FALSE |
| VNS2 < IVNS | TRUE | TRUE | TRUE |
| oR-tools > VNS1 | FALSE | FALSE | FALSE |
| oR-tools > VNS2 | FALSE | FALSE | FALSE |
| oR-tools > IVNS | FALSE | FALSE | FALSE |
| VNS1 > VNS2 | TRUE | TRUE | TRUE |
| VNS1 > IVNS | TRUE | TRUE | TRUE |
| VNS2 > IVNS | FALSE | FALSE | FALSE |

### 5.3.2 Experiment 2

In these experiments, we want to see on an algorithms, for example the IVNS, what is the best improvement pattern between a mixed movement strategy, that we will call mix (intra route and inter route movements are used all together in one only phase) or splitting in diversification and intensification phases, that we call di . The algorithm we are using is IVNS with the same parameters used in the Experiment 1. The only difference between the two taken algorithms is the parameter `improvement`, that has to respect the objective of this experiment. We use same pre-processing technique used in Experiment 1 to scale our simulations'data.

Running Wilcoxon rank test took us this results:
For the **solution value**:

- Two-tailed: We obtain a p-value for the Wilcoxon test equal to $2.577387199554837e-|145$, which is a very low value, even for significance levels $\alpha$ of very low order, so we can proceed with one-tailed tests.
- *One-tailed* (<0) : In the one-tailed we easily note a $p-value = 1.2886935997774184e-|145$ (very low) for the hypotyesis $H0$ " $di - mix \geq 0$ ", that can be rejected, so we can claim that di get better solutions than mix.

For the **computing times**:

- *Two-tailed*: We obtain a p-value for the Wilcoxon test equal to $3.629745791625171e-|228$, which is a very low value, even for significance levels $\alpha$ of very low order, so we can proceed with one-tailed tests.
- *One-tailed* (<0) :In the one-tailed we easily note a $p - value = 1.0$ (very high) for the hypotyesis $H0$ " $di - mix \geq 0$ ", that can't be rejected, so we have to do the hypotesis test for the opposite hypotesis.
- *One-tailed* (>0) :In the opposite $H0$ " $di - mix \leq 0$ " we have a p-value equal to $1.8148728958125854e - 228$, that is very low: we can claim that in this context the mix performs better than di in terms of computing times.

In this case, dividing two phases of search was better in terms of solution quality then mixing all neighborhood structures, but this involves two searchs, so it is more expensive in computational sense.

### 5.3.3 Experiment 3

In this experiment we want to test how introducing in our runs crossing-over operator can improve our solution. In this case, we will use IVNS algorithm with same parameters setting used in Experiment 1, but activating or deactivating the parameter `cross-over` and reducing the parameter `len_taboo`, that allows us to augment the cross-over operations that can happen in the algorithm. We will work in the same way as in the experiment 2, computing the Wilcoxon rank test to normalized solution values and computing times.

Test for **solution value**:

- Two-tailed: we obtain a p-value for the Wilcoxon test equal to $0.5258539548517263¦$ this is an high p-value, and it is saying to us that there aren't sifgnificant differences between having cross-over or not having in terms of quality of solution, so we can't stop here the analysis.

Test or the **computing times**:

- *Two-tailed*: we obtain a p-value for the Wilcoxon test equal to 0.23639872922467442 that, as before, is higher than statistically significant levels (maximum level as $\alpha = 0.1$). Also in this case, there aren't statistical differencies between using crossing over phase or not using it, and we stop the analysis.

These tests demonstrate the varying utility of different algorithmic modules. They highlight that the optimal choice, such as the inclusion of a crossing-over phase, should be assessed for each specific problem instance. This is because statistically significant differences may not always be present.

# Chapter 6

# Conclusions

The modular structure of our algorithms has allowed us to explore a wide range of algorithmic possibilities by creating new algorithms from our modules. Additionally, we conducted demonstrative analyses to illustrate how the behavior of a Single Solution Based solver can vary by modifying a few characteristics of the algorithm. Specifically, the modules we developed enabled us to implement the following algorithmic phases: improvement heuristics, improvement and shaking phases, destroy and repair phase, crossing-over phases, and route initialization. With these tools, we were able to construct several "hybrid" algorithms from classical single solution metaheuristics.

Clearly, compared to state-of-the-art solvers, our work is still in its infancy, as there are many areas for improvement in terms of computation and strategies. However, the utilization of these algorithmic structures certainly holds promise for enhancement by adding additional features. From what is evident in our work, there is potential for generalization using these tools to create new algorithms.

Furthermore, while our work focused on the CVRP, it is potentially generalizable to any type of VRP problem by making appropriate modifications to the calculation of the objective function and the study of constraints. For instance, extending the algorithm to the well-known CVRPTW problem is feasible with suitable adjustments.

In conclusion, our research lays the groundwork for further exploration and refinement of algorithmic solutions for VRP problems, offering avenues for future research and application in a broader context.

# Appendix A

# Code description

In our work we used **python language** to write the algorithms described in the last chapters. In particular, in this appendix we are going to describe briefly our code general informations.

## A.1 Classes

These classes are part of a solution framework for solving routing problems. Here's a breakdown of what each class does:

### Class Instance

Class Instance allows to manage and use in our solvers the CVRP instances.

- `__init__`: Initializes instance variables such as `maps` (coordinates matrix of clients), `demands` (list of demands for each customer), and $\texttt{v}_c apacities(listofvehiclecapacities$
$CalculatestheEuclideandistancematrixbetweenallpairsofpoints.$
- `compute_sol`: Computes a solution for the routing problem using our algorithms, and returns the solution as an instance of class `Solution`.
- `compute_ortSol`: Computes an alternative solution using the google's orTools solver (`orTSolution`) and returns it as an instance of class `orTSolution`.
- `plot_map`: Plots the map with the routes.

### Class Solution (inherits from Instance)

Class Solution is used to solve and manage solutions from our solvers.

- `__init__`: Initializes instance variables inherited from `Instance`, along with additional variables specific to a solution such as `routes`, `value`, `feasible`, and `time_execution`.

- `constraints`: Checks if the routes in the solution satisfy certain constraints.
- `standard_form_sol`: Converts the solution into a standard form (solution formulated with mixed-integer variables as in chapter 1).
- `route_form_sol`: Converts the solution into route form (solution given as a list of arrays containing the nodes indexes contained in the routes, as shown in the third chapter).
- `constraint_standard`: Checks if the solution satisfies standard constraints.
- `standard_form_solHigh`: Converts the solution into a specific standard form, possibly handling special or complex situations.
- `route_form_solHigh`: Converts the solution into a specific route form, possibly handling special or complex situations.
- `constraint_standardHigh`: Checks if the solution satisfies standard constraints with specific handling for special or complex situations.
- `plot_routes`: Plots the routes on the map, with an option to indicate the direction of the routes with arrows.

### The orTSolution Class

The orTSolution class is designed to solve the CVRP using the ORTools library.

- `get_solution()`: Solves the CVRP problem using ORTools, calculating the execution time.
- `constraints()`: Checks for any constraint violations in the solution.
- `standard_form_sol()`, `route_form_sol()`, `constraint_standard()`: Methods for manipulating the problem solution(similar to the other class `Solution`).
- `plot_routes()`: Visualizes the vehicle routes on a map.

The class offers an abstraction for solving CVRP problems using ORTools, providing methods to manipulate and analyze the obtained solution.

## The testAlgorithms Class

The testAlgorithms class is designed to execute and analyze different algorithms for solving a problem using the provided instance.

### Methods:

- `__init__(file)`: Initializes the class instance with a file path pointing to the problem instance. The file is used to create an instance of the problem.
- `executeTest(kind, par, reps)`: Executes a test with the specified kind of algorithm (`kind`), parameters (`par`), and number of repetitions (`reps`). It stores the results for further analysis.
- `export2dataFrame()`: Generates data frames containing information about the instance, solver, trials, and statistics of the results. This data can be exported for further analysis or visualization.
- `plot_result()`: Plots the results obtained from the executed tests, visualizing the routes.

The class provides functionality for testing and analyzing different algorithms' performance on the given problem instance.

## Algorithms

Moreover, our algorithms described in Chapter 3 are written in some modules:

- `clustering.py` and `sweepAlgorith.py`: these two classes are used to generate new solutions or cluster points (united by the function "first_route".
- `heuristics.py` for improvement heuristic deployments.
- `destroyRepair.py` : destroy and repair operations.
- `improvements.py`: improvements structures functions.
- `shakes.py`: shakes structures functions.
- `populationPhase.py`: crossing-over operations.
- `ClusterVNS.py`: this modules is the most general, where our solvers' algorithms are built and all the functions contained in the above modules are used in order to find the solutions.